On Learning Better Decision Trees

by

Muhammad Nauwar Al-Afandi

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

July, 1996

On Learning Better Decision Trees

BY

Muhammad Nauwar Al-Afandi

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

July 1996

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by

Muhammad Nauwar Al-Afandi

under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee:

Dr. Hussein Al – Muallim (Chairman)

Dr. Sulaiman Al – Bassam (Member)

Dr. Muhammad Al – Suwaiyel (Member)

Department Chairman

Dean, College of Graduate Studies

23-6-96
Date

This piece of work is presented to :

my wonderful and always giving parents

whose support, and prayers

led to this achievement,

my brother,

my loving wife,

and my lovely daughter *Einas*.

Acknowledgments

In the name of Allah, Most Gracious, Most Merciful. Read in the name of thy Lord and Cherisher, Who created. Created man from a {*leech-like* } clot. Read and thy Lord is Most Bountiful. He Who taught {the use of} the pen. Taught man that which he knew not. Nay, but man doth transgress all bounds. In that he looketh upon himself as self-sufficient. Verily, to thy Lord is the return {of all}.

(The Holy Quran, Surah 96)

First and foremost, all praise to Allah, *subhanahu-wa-ta'ala*, the Almighty, Who gave me an opportunity, courage and patience to carry out this work. I feel privileged to glorify His name in the sincerest way through this small accomplishment. I seek His mercy, favor, limitless help, guidance, and forgiveness. And I ask Him to accept my little effort. May He, *subhanahu-wa-ta'ala*, guide us and the whole humanity to the right path *(Amen)*. Peace and blessings of Allah be upon his prophet Muhammad.

Acknowledgment is due to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for providing support to this research work.

I would like to express my profound gratitude and appreciation to my Thesis Chairman, Dr. Hussein Al-Muallim, who helped me throughout this thesis and gave me a great deal of his valuable time. His continuous support and encouragement can never be forgotten.

I would also like to thank my Thesis Committee, Dr. Sulaiman Al-Bassam and Dr. Muhammad Al-Suwaiyel, for their consistent support and valuable suggestions.

I also wish to thank faculty, research assistants, graduate assistants, and the staff members of the Information and Computer Science Department for their support.

The encouragement and good wishes of my friends are also worthy of acknowledgment.

Finally, special thanks must be given to my parents for their encouragement and moral support.

Contents

List of Tables

List of Figures

Abstract

Name: Muhammad Nauwar Al-Afandi

Title: On Learning Better Decision Trees.

Major Field: Computer Science

Date of Degree: July, 1996

The ID3 algorithm follows a top-down approach to learn decision trees from examples. It generates sub-optimal trees because it is based on heuristics. In this thesis, we propose two approaches aimed at improving the quality of the generated decision tree in terms of the average classification cost to classify a case and the generalization performance of the tree. The first approach, called the weight adjustment approach, is based on the idea of assigning weights to the examples and adjusting these weights slowly in an iterative manner in order to alter the scores of the attributes in the attribute selection process so that a better decision tree is generated. The second approach is based on a new attribute selection criterion which combines the attribute's information-gain score (as used in ID3) computed from the training examples and the importance score related to the number of examples for which the attribute plays a role during classification by the previous tree. Judging from the experiments we conducted to test our approaches, the second approach added more significant improvement than the first one over ID3.

Master of Science Degree

King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

July, 1996

ملخص

الاسم : محمد نوار الأفندي

العنوان : بناء شجرات قرار أفضل

التخصص : علوم الحاسب الآلي.

تاريخ التخرج : تموز (يوليو) ١٩٩٦ م الموافق صفر ١٤١٧ هـ

تتبع خوارزمية "ID3" أسلوب التقسيم من الأعلى إلى الأسفل لبناء شجرات القرار من أمثلةٍ معطاة، وتولد الخوارزمية شجرات قرار محدودة المثالية كونها تعتمد على إحصاءات تقديرية في عملية توليد الشجرات.

نعرض في هذه الدراسة أسلوبين جديدين لتحسين مستوى جودة الشجرة المبنية عن طريق تقليل معدل كلفة تصنيف الأمثلة وزيادة مستوى دقة تصنيف الحالات الجديدة. ويسمى الأسلوب الأول أسلوبَ تعديل الأوزان بينما يسمى الثاني أسلوبَ التطوير المتتابع.

يقوم الأسلوب الأول على أساس إعطاء وزن لكل مثال بقيمةٍ حقيقيةٍ موجبة، ومن ثم تعديل هذا الوزن بشكل متكرر وبطيء حيث تتغير القيم التي تحصل عليها السمات أثناء اختيارها، وبالتالي يتم بناء شجرة قرار أعلى جودة. ومن ناحيةٍ أخرى، يقوم الأسلوب الثاني على أساس استخدام طريقةٍ جديدة لاختيار السمات بحيث يجمع بين فوائد استخدام قيمة الفائدة المعلوماتية المستخدمة في خوارزمية "ID3" والمحسوبة من الأمثلة مع قيمة الأهمية لكل سمة والتي تعتمد على عدد الأمثلة التي تصنفها السمة في الشجرة السابقة.

وتشير نتائج التجارب التي تم إجراؤها على كلا الأسلوبين إلى أن أسلوب التطوير المتتابع قد حقق نتائج أكثر أهميةً من أسلوب تعديل الأوزان مقارنة مع خوارزمية "ID3".

درجة ماجستير في العلوم

جامعة الملك فهد للبترول و المعادن

الظهران، المملكة العربية السعودية

تموز (يوليو) ١٩٩٦ الموافق صفر ١٤١٧

Chapter 1

Introduction

One of the most striking differences between how people and computers work is that humans, while performing any kind of activity, usually simultaneously expand efforts to improve the way they perform it. This is to say that human performance of any task is inseprabably interwined with a learning process, while computers are typically only executors of procedures supplied to them. They may execute very efficiently, but they do not self-improve with experience.

Research in machine learning has been concerned with building computer programs able to construct new knowledge or to improve already possessed knowledge by using input information. So far, this input information (examples, facts, descriptions, etc.) has been typically typed by a human instructor. Future machine learning

programs will undoubtedly be able to receive inputs directly from the environment through a variety of sensory devices [Fay91].

In this thesis, we will restrict our attention to the task of learning classification rules from previous cases or examples. Consider the simple task of deciding on a certain day whether to play soccer or not. Let us look at how a human would construct a decision rule for this task. First, he would recall previous experiences regarding this task. Then, he would study those experiences trying to detect the factors that led to the decision at each experience. These factors or properties could involve the day outlook, which might be sunny, overcast, or rainy, the temperature, the rate of humidity, and possibly whether it was windy or not. Table 1.1 shows two previous experiences with four properties of the weather along with the decision made in each case. Based on the decisions made at those experiences, he would then construct the desired rule that can be applied to make a decision for future cases. The task of deciding whether to play or not can be thought of as a *classification*

Outlook	Temp(F)	Humidity	Windy ?	Decision
sunny	75	normal	true	Play
sunny	80	high	true	Don't Play

Table 1.1: Two previous experiences each recording the values of 4 properties and a decision.

task. Classification means assigning objects to categories or classes based on their properties. In the context of deciding whether to play soccer or not, the two decisions

can be thought of as two classes. Each experience represents a pre-classified case to be studied in order to make decisions in new situations. In this thesis, we will refer to experiences as *cases* or *examples* and to properties as *attributes*.

One may think of implementing the desirable classifier by manually writing a program for it. For example, a classifier may be represented as nested if or case statements so that new cases are matched against the conditions of these statements in order to make decisions. However, as the number of properties and the number of previous cases get larger, the program becomes more expensive and more difficult to construct, maintain, or debug. Imagine having an application with thousands of cases and tens of both relevant and irrelevant properties to be searched using nested if or case statements. Obviously, it would be impractical to attempt to handle the classifier construction task through manual means in such situations.

The goal of machine learning is to automate the process of building classifiers. The key idea is to summarize large sets of cases by capturing compact patterns and statistics existing in them and using these to construct the target classifier. Thus, as shown in Figure 1.1, the input to a learning algorithm is a set of cases or examples and the output is a classifier represented in some appropriate language.

The most well-known approach to learn classifiers out of cases is to represent the classifier as a decision tree and follow a top-down approach to construct such a

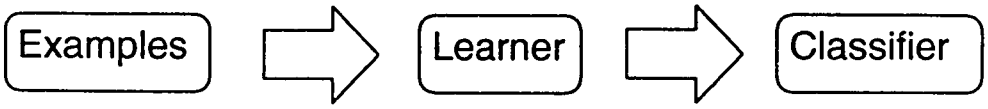


Figure 1.1: The scenario of learning.

tree. ID3 [Qui93, Qui86] is a famous algorithm that follows the top-down approach in building a classifier. This algorithm has been applied successfully to a variety of real-world problems. A major feature of this algorithm is its efficiency. ID3 is a fast algorithm capable of generating a classifier for applications involving tens of attributes and hundreds of cases in less than a minute on a moderate computer. However, ID3 is based on heuristics, and thus, it typically gives only sub-optimal classifiers. In real-world applications, one might be willing to spend hours and possibly days of CPU time in order to get better classifiers that suit specific requirements, e.g. give more accurate and/or economical decisions. The goal of this thesis is mainly to trade CPU time for the sake of generating better decision trees than those generated by the ID3 algorithm.

Chapter 2 defines decision trees as structures used to store the learned classifiers and discusses why they are more interesting than other forms of classifiers. It also explains the top-down approach and focuses the light on important criteria used to evaluate classifiers. Finally, the chapter presents the C4.5 package as the state of the art implementation of the ID3 algorithm of Quinlan, and surveys some attempted extensions of this algorithm.

Chapter 3 describes an important weakness of the top-down approach, namely, the locality of decisions.

The succeeding two chapters, 4 and 5, propose two methods to improve the top-down approach in order to build better classifiers. Chapter 4 explains our iterative approach which is based on the idea of assigning weights to cases and adjusting these weights in order to get a better classifier with respect to a certain evaluation criteria. Chapter 5 presents our successive refinement approach as a method to learn from mistakes and improve the way we build classifiers by utilizing the knowledge we got from previously generated classifiers.

Chapters 4 and 5 include the results of experiments we have performed on both artificial and natural domains. The data in the domains come from the machine learning database of the University of California at Irvine, which is a widely used source for benchmark data in machine learning research.

Chapter 6, the conclusion of this thesis, includes a summary of the achievements we got from our work and recommendations we suggest for future research.

Chapter 2

Background

2.1 Popular Approaches to Learning from Examples

Classifiers learnt from examples can be represented in many forms. The popular ones include sets of classification rules, neural networks, and decision trees. In this thesis, our attention will be limited to the latter form of classifiers; that is decision trees. We will, however, discuss the former two forms of classifiers briefly here.

2.1.1 Classification Rules

A classification rule for making a decision consists of a specification of the values of one or more properties on the left hand side and that decision on the right hand side. A simple rule consistent with the examples of Table 1.1 may be:

IF (Outlook = sunny and Humidity = normal) THEN the deicsion is to *play.*

Learning classifiers as a set of rules is not easy because obtaining a small and consistent set of rules to be used in classifying cases such that one rule will match any single case is a difficult task [Fay91, Qui93].

Several algorithms have been proposed to learn classifiers as sets of rules. For example, CN2 was develpoed to combine the efficiency and ability to cope with noisy data of ID3 with the if-then rule form and flexible search strategy of the AQ family. The representation for rules output by CN2 is an ordered set of if-then rules, also known as a *decision list* [CN89].

2.1.2 Neural networks

In its modern form, a neural network consists of units connected by links, as illustrated in Figure 2.1. Three kinds of units are distinguished: input units, such as

A and B, that introduce information from the environment to the network, output units, such as E, that give the result; and all other units, such as C and D, that are "hidden" from the environment. Each link has an associated weight and some units have a bias that appears below the unit. To process a case, the input units are first assigned numbers between 0 and 1 representing the attribute values. Each unit's input I is determined as the sum of the weighted output of the units connected into it, plus the unit bias, and the unit's output determined as

$$\frac{1}{e^{-I}+1}$$

which ranges from 0 to 1 as I ranges from $-\infty$ to ∞.

For example, if unit A had the value 0 while unit B has the value 1, the input to unit C would be $0 \times 5.7 + 1 \times 5.7 + (-2.2) = 3.5$ and its output would be 0.97. This output in turn becomes an input to unit E, with weight 6.5.

The values of the network weights and biases are learned through repeated examination of the cases. The deviation of each output unit's output from its correct value for the case is *back-propagated* through the network, where all relevant connection weights and unit biases are adjusted, using gradient descent, to make the actual output closer to the target. Training continues until the weights and biases stabilize [LV91].

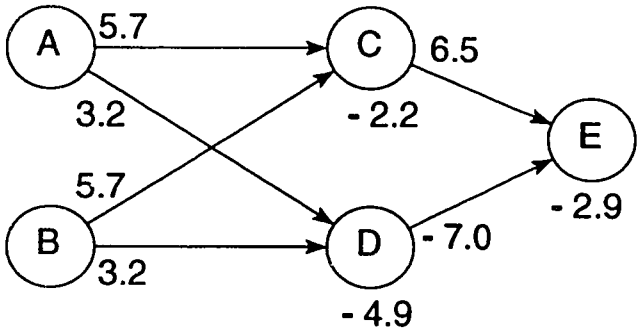


Figure 2.1: A neural network for XOR.

Neural networks may be considered as strictly "black box" classifiers. Decision trees and symbolic rule classifier approaches produce classifiers in a symbolic logical format that is intended to be meaningful to humans. However, neural networks are essentially a "curve fitting" technique where the target language in which the classifier is expressed is intended to be strictly an internal representation for which no "understandable" interpretation need to be found. Moreover, neural networks require large amount of time to be trained. Cases are typically iterated through the network thousands of times before the network converges on a local minimum. In fact, there is a significant evidence that the training of most common types of networks is an NP-hard problem [BR88].

Furthermore, there is an important network design problem that remains open: how many internal layers/units should a network have for a given learning task? If this number is too large, the network will simply rote learn the training set and no induction will take place. If it is too small, the network may never be able to

converge on a solution that is consistent with the set of cases. This is just one of the many parameters that a user needs to specify such as: how to represent the problem (input layer)? how to set the learning rate parameter? how to initialize the connection weights? etc [LV91].

To summarize, neural networks possess a powerful representation and curve fitting capability resulting in increased complexity of the learning process. The fact that the search space for possible solutions is larger may increase the chance of stumbling on a "better" solution, but is certain to increase the chance of landing on a "bad" one as well. Neural networks also come with a host of open design questions, such as the number of nodes to use in the hidden layer. The reliance on many user-specified parameters as well as the initial settings problem are factors that make a decision tree approach more practical than a neural network approach for industrial applications of machine learning [LV91, BR88].

2.1.3 Decision Trees

Informally, decision trees consist of three components: decision nodes, branches representing outcomes of decision nodes, and leaves. Let A be a set of attributes and C be a set of classes. A decision tree is a tree structure with the following properties:

- Each nonterminal node, called a decision node, is labelled with a test involving one of the attributes in A. The test has a finite number of disjoint outcomes.
- Each outgoing branch from a nonterminal node corresponds to one of the outcomes of the test at the node.
- Each terminal (leaf) node is labelled with one of the classes belonging to C.

Each leaf in the tree represents a class (classification rule). The conjunction of tests on the branches from the root to that leaf constitutes the preconditions of that rule. This is why a decision tree can be thought of as a set of rules, where no two rules can match the same example [Fay91, Qui93].

Figure 2.2 shows a decision tree which is a possible classifier for the soccer problem described earlier in Chapter 1. Both cases in Table 1.1 can be easily classified by this decision tree. For instance, in order to classify the first case with this tree, we need to apply the test at the root node first. Since the value of the attribute "outlook" is sunny, the classification process should select the branch labeled sunny. Next, we test the value of the attribute "humidity". As the case has the value normal for this attribute, attention switches to the branch labeled normal. Finally, the classification process ends with the leaf labeled P (for Play), and the label is returned as the class of the case.

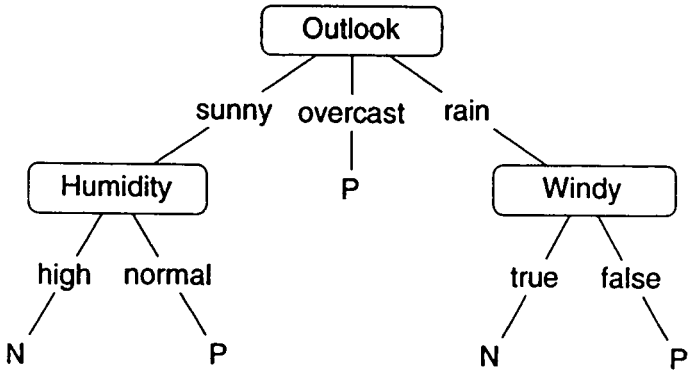


Figure 2.2: A simple decision tree.

Automatic Construction of decision trees from examples is possible as we already have algorithms that can generate them, such as C4.5 [Qui93] and CART [BFOS84]. Both packages follow the top-down approach to build decision trees. This approach will be described later in this chapter.

Decision trees represent only one approach to the concept learning problem. What makes them preferable over other approaches, such as neural networks, or pattern recognition based approaches are the following factors:

1. Decision trees offer an efficient means for processing large data sets of cases [FI92]. This is because the learning problem is being partitioned among branches into sub-problems with smaller sets of training data [Fay91].

2. Decision trees can be transformed into symbolic classification rules, as done in C4.5. These rules are symbolic since they are expressed as simple high-

level conditions, and therefore, they are not difficult for domain experts to understand [FI92, Fay91].

3. Unlike neural networks, no extra information is required other than what is provided in the set of cases as parameters to the decision tree generation approach [Fay91].

Looking at these factors, the decision trees approach has the potential for being a powerful and flexible classification tool. However, one can point out several drawbacks associated with the above advantages of decision trees. For example, partitioning the set of cases might lead to problems like loss of information or ending up with decisions based on small number of cases which means less confidence in these decisions in the process of building the tree. Furthermore, having a symbolic set of classification rules gives credit to understandability but not accuracy which could be the only important criteria. From the industrial applications point of view, things like efficiency, and convenience are very essential [Fay91, CN89].

2.2 Trading CPU Time for Quality in Decision Tree Learning

The space of decision trees is large and an exhaustive search approach is intractable. Finding optimal decision trees with respect to a certain evaluation criteria is an NP-hard problem [HR76]. This justifies the use of heuristics to help us limit our search and obtain good decision trees in an acceptable amount of time. Previous algorithms such as ID3 [Qui93] and CART [BFOS84] are examples of this. These algorithms are fast in generating decision trees because they are based on a simple greedy technique with no lookahead or backtracking.

However, given the recent advances in computer hardware technology, more computations can now be carried out at lower costs. Consequently, one may be interested to spend more CPU time and obtain better decision trees than those provided by the rapid algorithms ID3 and CART. In this work, we aim at trading some bounded computation time for the sake of getting better decision trees. Our proposed approaches are aimed at generating better decision trees than those generated by C4.5 but within practical execution time.

Before discussing our new approaches, we will devote the rest of this chapter and the next chapter for reviewing the conventional approach to decision tree learning

and limitations of that approach.

2.3 The Top-Down Approach

Given a set of cases S and a set of classes $\{C_1, C_2, \cdots, C_k\}$, a decision tree can be generated by applying the following steps:

1. If S contains one or more examples, all belonging to the same class, then the decision tree for S is a leaf labeled with that class.

2. If S contains no examples, the decision tree is again a leaf labeled using information other than S. For instance, an overall majority class could be used as a default class based on background knowledge of the domain.

3. If S contains examples belonging to different classes, then S is partitioned based on an attribute chosen according to some criteria. If attribute A with values $\{V_1, V_2, \cdots, V_k\}$ was selected to form the test, it will split S into $\{S_1, S_2, \cdots, S_k\}$ where the value of attribute A in each example in S_i is V_i. As a result, we will have a decision node with a test labelled with A and one subtree T_i for each outcome of this test such that the value of attribute A of each example in S_i is V_i. The algorithm is recursively applied at each subtree T_i on its S_i set of cases [Qui93] as illustrated in Figure 2.3.

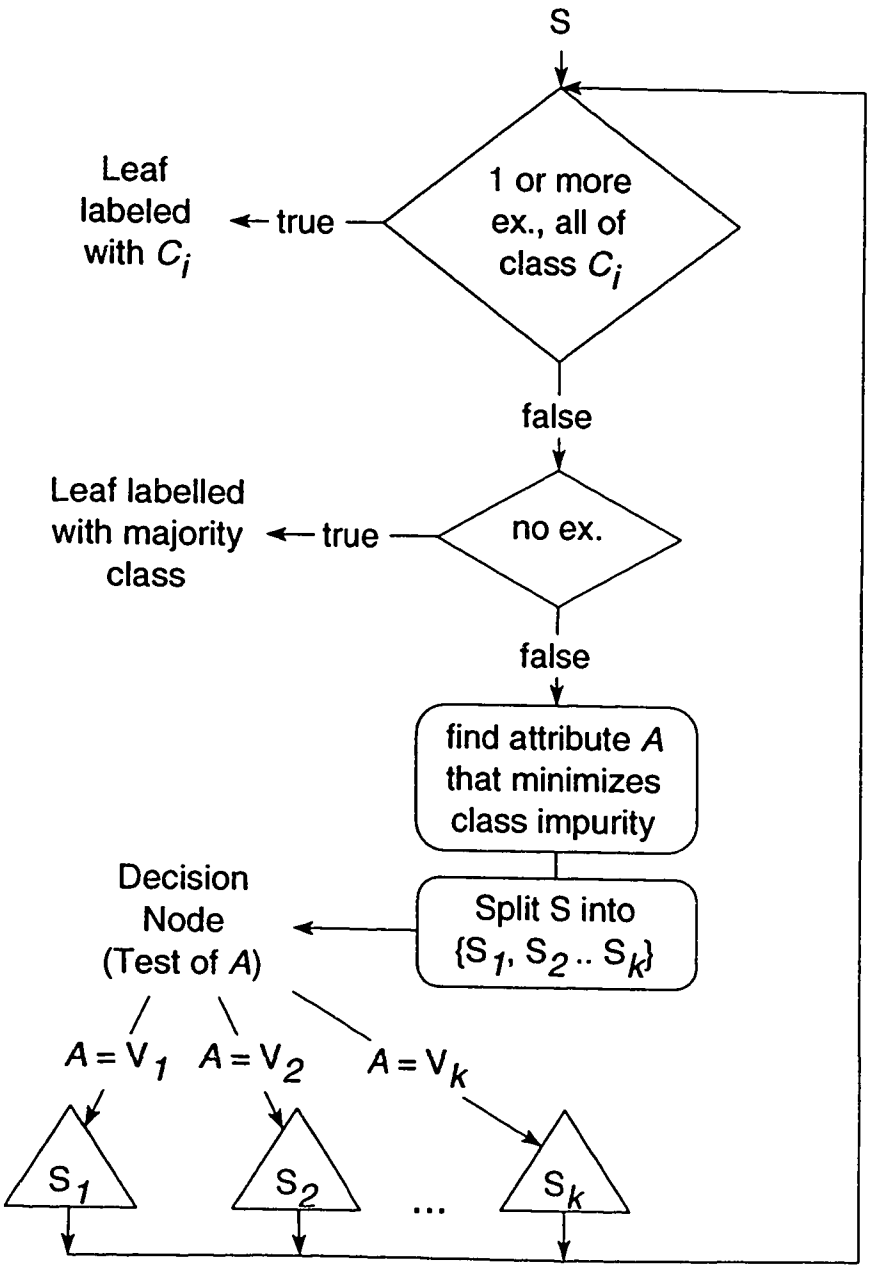


Figure 2.3: The Top-Down approach.

2.3.1 Illustration

Let us try to apply the top-down approach on the soccer domain mentioned before in order to construct a decision tree from a set of old cases. From this point on, we will refer to cases used in constructing the decision tree as the training set and cases reserved to estimate the generalization performance (i.e. the prediction power) of the tree as the test set.

Given a small training set like the one in Table 2.1 (copied from [Qui93]) in which there are four attributes and two classes, the top-down approach divides the set of training set successively until all the subsets consist of cases belonging to a single class. Obviously, the cases in Table 2.1 do not belong to the same class. Therefore, the top-down approach tries to split them into subsets. Suppose the chosen test was on attribute "outlook" with three outcomes, outlook = sunny, outlook = overcast, and outlook = rain. Figure 2.4 shows the resulting partial decision tree after splitting the set of cases into three subsets.

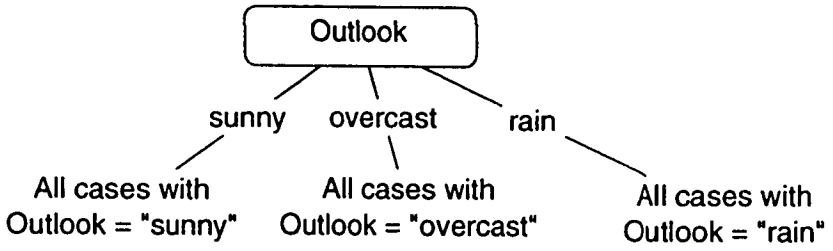


Figure 2.4: The resulting partial decision tree after first partitioning.

Outlook	Temp(F)	Humidity(%)	Windy ?	Class
sunny	75	70	true	Play
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	68	false	Play

Table 2.1: A table of data where all examples are classified correctly.

The middle subset contains cases of class Play only, but the first and the third subsets have mixed classes. If the first subset gets further divided by a test on humidity, with outcomes humidity ≤ 75 and humidity > 75, and the third subset by a test on windy, with outcomes windy = true and windy = false, each of the subsets would contain cases from a single class. The final divisions of the subsets is illustrated in Figure 2.5. The The corresponding decision tree is shown in Figure 2.6.

Partition of cases:

outlook = sunny

humidity $\leq$ 75

Outlook	Temp	Humidity(%)	Windy?	Decision
sunny	75	70	true	Play
sunny	69	70	false	Play

humidity > 75

Outlook	Temp	Humidity(%)	Windy?	Decision
sunny	80	90	true	Don't Play
sunny	85	85	false	Don't Play
sunny	72	95	false	Don't Play

outlook = overcast

Outlook	Temp	Humidity(%)	Windy?	Decision
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play

outlook = rain

windy = true

Outlook	Temp	Humidity(%)	Windy?	Decision
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play

windy = false

Outlook	Temp	Humidity(%)	Windy?	Decision
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Figure 2.5: The final partition of cases.

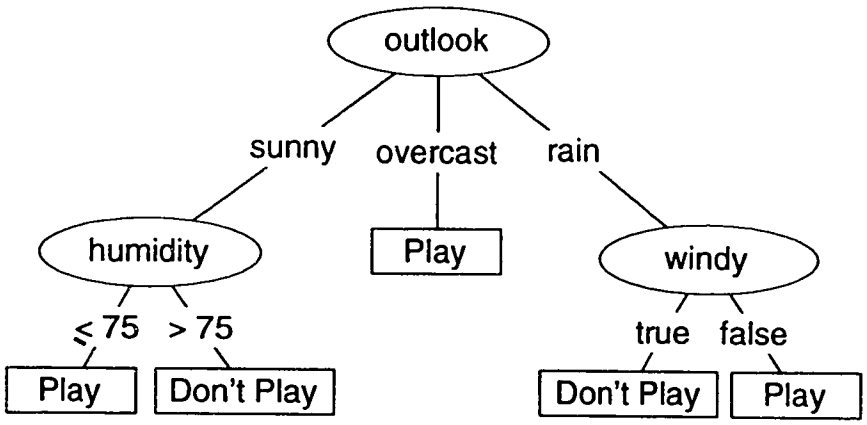


Figure 2.6: The corresponding decision tree of the partitions.

2.4 Attribute Selection

The tree generated by the top-down approach depends on the choice of tests at each recursive call. From the same table of cases as Table 2.1, many consistent decision trees can be constructed using different choice of tests. Different criteria for the choice of tests result in different decision trees. Figure 2.7 shows another tree generated from the same set of cases used to generate the tree of Figure 2.6 by the top-down approach using a different criterion for the choice of tests.

Both trees in Figures 2.6 and 2.7 are indeed consistent with the training cases of Table 2.1, but which one is better? In the tree building process, any test that divides the set of cases S in a nontrivial way, such that at least two of the subsets

$\{S_i\}$ are not empty, will eventually result in a partition into single-class subsets (all cases belonging to a single-class subset are of the same class), even if all or most of the subsets contain a single case. However, the tree building process is intended to build a tree that reveals the structure of the domain and so has predictive power.

For that, we need a significant number of cases at each leaf. In other words, we want a partition that has as few divisions as possible. Therefore, more compact trees will be favoured over larger trees. In fact, with compact trees, as more cases are used to choose a test at each recursive call, the confidence in this choice increases, so does the predictive power [Qui86].

If we choose a test based on a small number of cases, then it is possible that there was a bias toward the chosen test and, in effect, a better test might have been dropped out due to this bias. This will result in a tree that does not necessarily reveal the structure of the domain and, hence, have less prediction power over new cases. On the other hand, with more compact trees having more confidence in the choice of tests, the classes of new cases are expected to be predicted more correctly. Compare the partitioning of cases in Figure 2.5 and the corresponding decision tree in Figure 2.6 with the one in Figure 2.8 and the corresponding tree in Figure 2.7.

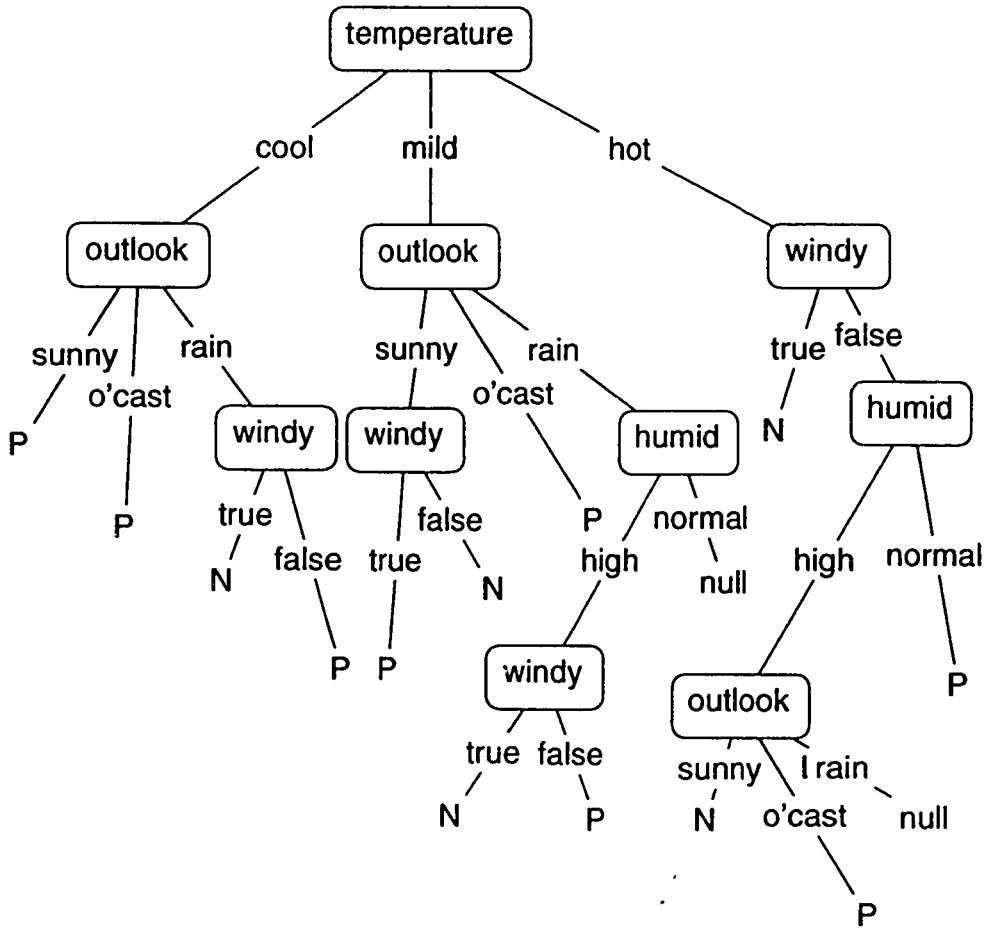


Figure 2.7: A more costly and more complex decision tree. Cool scale temperature corresponds to temperatures $< 70F$. Mild scale temperature corresponds to temperatures $\geq 70F$ and $< 80F$. Hot scale temperature corresponds to temperatures $\geq 80F$. Humidities less than or equal to 75% are considered normal. Otherwise, they are considered high.

temperature < 70
| outlook = sunny
| | Outlook | Temp | Humidity(%) | Windy? | Decision
| | sunny | 69 | 70 | false | Play
| outlook = overcast
| | overcast | 64 | 65 | true | Play
| outlook = rain
| | windy = true
| | | rain | 65 | 70 | true | Don't Play
| | windy = false
| | | rain | 68 | 80 | false | Play
temperature ≥ 70 and < 80
| outlook = sunny
| | windy = true
| | | sunny | 75 | 70 | true | Play
| | windy = false
| | | sunny | 72 | 95 | false | Don't Play
| outlook = overcast
| | overcast | 72 | 90 | true | Play
| outlook = rain
| | humid > 75
| | | windy = true
| | | | rain | 71 | 80 | true | Don't Play
| | | windy = false
| | | | rain | 75 | 80 | false | Play
| | | | rain | 70 | 96 | false | Play
| | humid ≤ 75
| | | null

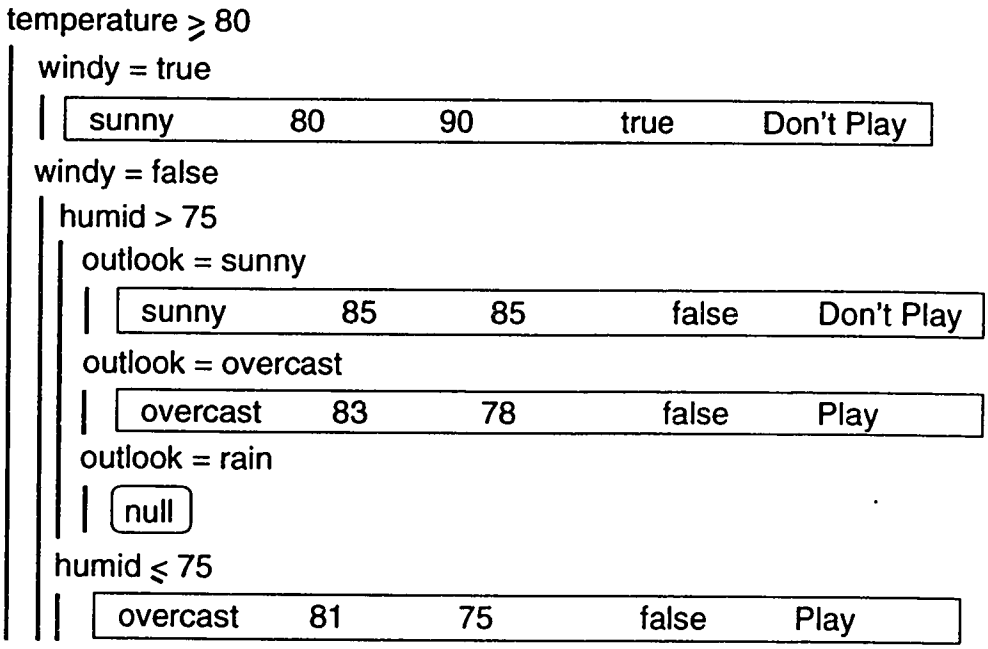


Figure 2.8: The partition of cases resulting in a complex tree.

The top-down decision tree building method is usually implemented in a non-backtracking, greedy fashion. Once a test has been selected to partition the current set of cases, usually on the basis of maximizing some local measure of progress, the choice is set and the consequences of alternative choices are not explored. This makes it necessary to choose the test carefully. With most data sets, if attributes are selected randomly, classification performance of the generated decision trees tends to be poorer than that of trees generated with a careful attribute selection criteria [LW94].

Suppose we have a possible test with n outcomes that partitions the set S of cases into subsets $\{S_1, S_2, \cdots, S_n\}$. If this test is to be evaluated without exploring subsequent divisions of the S_i's, the only information available for guidance is the distribution of classes in S and its subsets. The goal is to build a compact tree by choosing tests which seem to minimize the class impurity in the subsets if we partition on the test. This suggests choosing the test which minimizes the class impurity in each subset of cases resulting from partitioning on this test.

For example, in the soccer domain, we noticed that choosing the attribute "outlook" as the test on the root of the tree generates a more compact tree than choosing "temperature" at the root because "outlook" leads to less impurity as shown in Figure 2.9. Judging from the class distribution shown in Figure 2.9 and taking the average impurity over the subsets, attribute "outlook" is clearly a better choice because it minimizes impurity.

Obviously, we need a measure of impurity that indicates how good is choosing a certain test over another. In the next section, two popular measures are explained, namely, the information-gain and the gain-ratio.

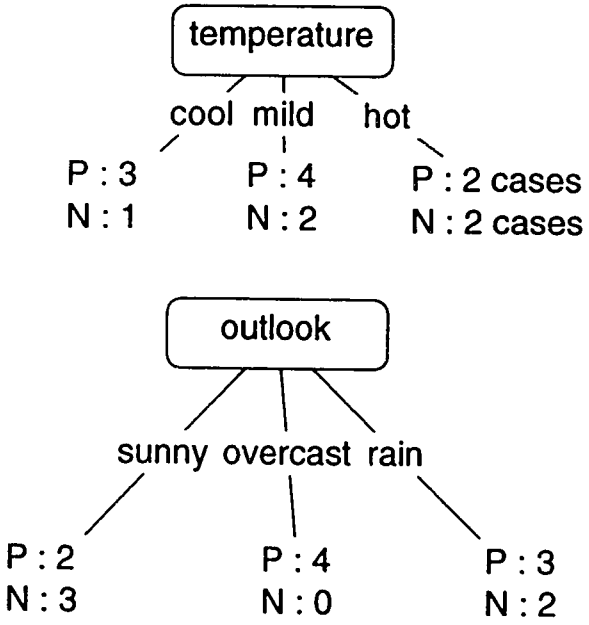


Figure 2.9: The class distribution on "outlook" and "temperature".

2.5 The Information-Gain and Gain-Ratio Measures

For attribute selection during the decision tree generation process, the ID3 algorithm as of the most recent implementation, C4.5 package, uses a criterion called *gain* which is an information-based method.

2.5.1 The Information-Gain

One of the popular measures used to measure the impurity is the information-gain. The information theory underlining this criterion states the following: the information conveyed by a message depends on its probability and can be measured in bits as minus the logarithm to base 2 of that probability. For example, if there are eight equally probable messages, the information conveyed by any one of them is $-\log_2 \frac{1}{8}$ or 3 bits.

Let S be a set of cases (cases used to generate the tree are called training cases), and let $freq(C_i, S)$ stand for the number of cases in S that belong to class C_i. We will use the notation $|\ S\ |$ to denote the number of cases in the set S. For simplicity, we will consider a domain with two classes, P and N.

If we pick a case from the set S at random and say that it belongs to some class C_j. This message has probability $\frac{freq(C_j,S)}{|S|}$ and so the information it conveys is

$$-\log_2 \frac{freq(C_j,S)}{|S|} \quad \text{bits.}$$

The expected information from this case is calculated by summing over the classes

in proportion to their frequencies yielding:

$$info(S) = -\sum_{i=1}^{k} \frac{freq(C_j, S)}{\mid S \mid} \log_2 \left(\frac{freq(C_j, S)}{\mid S \mid} \right) \text{ bits.} \qquad (2.1)$$

This measure gives the average amount of information needed to classify (identify the class of) a case in S and is also known as the *entropy* of the set S.

Now let us assume that the set S has been partitioned based on a test X with n outcomes. The expected information requirement can be found as the weighted sum over the subsets, as

$$info_X(S) = -\sum_{i=1}^{n} \frac{\mid S_i \mid}{\mid S \mid} \times info(S_i). \qquad (2.2)$$

Therefore, the information-gain is

$$Information\ Gain(X) = info(S) - info_X(S) \qquad (2.3)$$

The information-gain measure indicates the information gained by partitioning S on the test X. It chooses the test which maximizes the information gain [Qui93].

To illustrate the idea behind the information-gain measure, let us go back to the soccer domain. There are two classes, nine cases belonging to class Play and five belonging to class Don't Play. Therefore,

$$info(S) = -\tfrac{9}{14} \times \log_2(\tfrac{9}{14}) - \tfrac{5}{14} \times \log_2(\tfrac{5}{14}) = 0.940 \text{ bits.}$$

The expected information requirement due to a test on "outlook" as in Figure 2.6 is

$$info_X(S) = \frac{5}{14} \times (-\frac{2}{5} \times \log_2(\frac{2}{5}) - \frac{3}{5} \times \log_2(\frac{3}{5})) + \frac{4}{14} \times (-\frac{4}{4} \times \log_2(\frac{4}{4}) - \frac{0}{4} \times \log_2(\frac{0}{4})) + \frac{5}{14} \times (-\frac{3}{5} \times \log_2(\frac{3}{5}) - \frac{2}{5} \times \log_2(\frac{2}{5})) = 0.694 \text{ bits.}$$

The information-gain by this test is, in turn, $0.940 - 0.694 = 0.246$ bits.

Now let us evaluate the information gained by dividing the set S on the attribute "temperature" as in Figure 2.7. The expected information requirement is

$$info_X(S) = \frac{4}{14} \times (-\frac{3}{4} \times \log_2(\frac{3}{4}) - \frac{1}{4} \times \log_2(\frac{1}{4})) + \frac{6}{14} \times (-\frac{4}{6} \times \log_2(\frac{4}{6}) - \frac{2}{6} \times \log_2(\frac{2}{6})) + \frac{4}{14} \times (-\frac{2}{4} \times \log_2(\frac{2}{4}) - \frac{2}{4} \times \log_2(\frac{2}{4})) = 0.910 \text{ bits.}$$

The information-gain by this test is, in turn, $0.940 - 0.910 = 0.030$ bits. This gain is less than the gain resulting from the test on "outlook." The information-gain measure would then prefer the test on "outlook" over the latter test on "temperature" [Qui86, Qui93, FI92].

2.5.2 The Gain-Ratio

One serious deficiency in the information-gain measure is its bias favouring tests with many outcomes [Qui93]. For example, consider a training set of cases from a hypothetical medical diagnosis classification domain. Let one of the attributes be the patient identification, which is usually unique. If this attribute is to be used in partitioning, we will end up with a large number of partitioned subsets each of which contains only one patient example. Since all of these one-case subsets necessarily contain cases of a single class, $info_X(S) = 0$, and the information gain from using this attribute to partition the set of training cases becomes maximal. However, this partitioning is useless for predicting new cases, especially if we have a large number of cases with unique ID's where it would be very costly to store the tree in terms of space [Qui93, FI92].

To straighten this bias, the information-gain measure has to be adjusted by a kind of normalization in which the apparent gain of attributes with many outcomes is reduced. This is done by considering the outcome of the test to classify a case instead of the class to which the case belongs. By analogy with the definition of $info(S)$, we have

$$Split\ info(X) = -\sum_{i=1}^{n} \left(\frac{|\ S_i\ |}{|\ S\ |} \log_2 \frac{|\ S_i\ |}{|\ S\ |} \right) \tag{2.4}$$

While the information-gain measures the information relevant to classification that arises from partitioning the set S into n subsets, the *Split info* represents the potential information generated by the same division. The gain-ratio is therefore

$$Gain\ Ratio(X) = \frac{gain(X)}{Split\ info(X)} \tag{2.5}$$

The gain-ratio measures the proportion of information generated by a useful split, i.e., a split that seems to help classification. It selects the test which maximizes the ratio above [Qui93].

If the split is useless, such as selecting the patient ID, the attribute will not be ranked highly by the gain ratio criterion. If there are v classes and n cases in the training set, then the information-gain will be $\log_2(v)$ at most and the *Split info*(A) will be $\log_2(n)$ since every case has a unique outcome. It would be reasonable to assume that the number of cases is much larger than the number of classes such that the ratio would have a small value [Qui93].

To illustrate the idea behind the gain-ratio measure, we go back to the soccer domain. The test on attribute "outlook" splits the subset S into three subsets containing five, four, five cases respectively as in Figure 2.9. The split information

is calculated as

$$-\frac{5}{14} \times \log_2(\frac{5}{14}) - \frac{4}{14} \times \log_2(\frac{4}{14}) - \frac{5}{14} \times \log_2(\frac{5}{14}) = 1.577 \text{ bits.}$$

For this test whose information-gain is 0.246 (as calculated before), the gain-ratio is $\frac{0.246}{1.577} = 0.156$. For the test on attribute "temperature", the split information is equal to

$$-\frac{4}{14} \times \log_2(\frac{4}{14}) - \frac{6}{14} \times \log_2(\frac{6}{14}) - \frac{4}{14} \times \log_2(\frac{4}{14}) = 1.557 \text{ bits.}$$

The information-gain is 0.030. Therefore, the gain-ratio is $\frac{0.030}{1.557} = 0.019$. The gain-ratio measure would also prefer the test on "outlook" over the latter test on "temperature" [Qui86, Qui93, FI92]

According to (Quinlan, 1993), the gain-ratio criterion is robust and usually selects better test than those selected by the information-gain criterion. When the tests are all binary and the difference between them is in the proportions of examples in each outcome, the gain-ratio criterion is preferred. On the other hand, though it generates smaller trees, it is biased towards unbalanced splits where one subset S_i is far smaller than others.

2.6 Evaluation of Decision Trees

Since our objective is to learn improved decision trees, we need to make precise what we exactly mean by a "better" decision tree. Several measures have been used in the machine learning literarure to assess the quality of a decision tree. Among those measures are the number of nodes, the number of leaves, the average depth of the tree, the maximum depth of the tree, the average cost to classify a case (the average number of tests required to classify the case), and the generalization performance (prediction power over new cases not used in the training set). Our interest will be focused on the last two measures because, in practice, they are more important than the remaining ones. In other words, we will use the average classification cost and the generalization performance criteria to evaluate the quality of decision trees.

2.6.1 Average Classification Cost

Under the criterion of average classification cost (or average cost, for short), we measure how much it costs, on average, to classify a case using a given decision tree. This cost comes, in practice, from the tests that must be performed before the class of a certain case is determined. In real life, those tests could be actual test procedures with money involved. For instance, in medical domains, a test may involve costly procedures such as blood test, X rays, etc. In this work, we will

consider (for simplicity) the case where all tests have uniform cost. Thus, the less average number of tests required to predict the class of a case, the better the tree.

In such a setting, the average cost is estimated as follows. We sum up the number of tests required to classify each case in the training set and divide the sum by the number of cases in the set.

To illustrate the importance behind the average cost, let us go back to the soccer domain. The decision tree in Figure 2.6, which is more compact, has an average cost of

$$AverageCost(T_1) = \frac{(2\times2+3\times2)+(4\times1)+(2\times2+3\times2)}{14} = 1.71 \text{ tests.}$$

On the other hand, the average cost for the tree in Figure 2.7 is

$$AverageCost(T_2) = \frac{(2+2+(3+3))+((3+3)+2+((4+2\times4)))+(2+((4+4)+3))}{14}$$
$$= 3.07 \text{ tests.}$$

Obviously, the compact tree in Figure 2.6 has less average cost which makes it more economical and, hence, preferable.

2.6.2 Generalization Performance

The second criteria of interest to us is the generalization performance. It indicates how accurately a decision tree can classify new cases (called unseen cases) which were not used in generating the tree. The aim is to minimize the number of cases which are miss-classified by the tree, hence maximizing the prediction power, and improving the generalization performance [Qui93]. Later in this chapter, we will see how C4.5 estimates the generalization performance of decision trees.

2.7 The C4.5 Package

C4.5 is a commercially available software package recently released by Quinlan. This package includes the most recent implementation of the ID3 algorithm in addition to many other features. In this section, we describe two features of C4.5 that are used heavily in the experimental work of this thesis. These features are pruning and generalization performance estimation.

2.7.1 Pruning

During the process of building the tree, the recursive partitioning method of constructing decision tress continues to subdivide the set of training cases until each subset in the partition contains cases of a single class, or until no test offers any improvement. The result is often a very complex tree that "overfits the data" by inferring more structure than is justified by the training cases [Qui93].

There are basically two ways in which the recursive partitioning method can be modified to produce simpler trees: deciding not to divide a set of training cases any further, or removing retrospectively some of the structure built by recursive partitioning [Qui93, EMS93].

The former approach, usually called *stopping* or *prepruning*, has the attraction that time is not wasted assembling structure that is not used in the final simplified tree. The typical approach is to look at the best way of splitting a subset and to assess the split from the point of view of statistical significance, information gain, error reduction, or any other criterion. If this assessment falls below some threshold, the division is rejected and the tree for the subset is just the most appropriate leaf. However, as Breiman *et al.* [BFOS84] point out, such stopping rules are not easy to get right — too high a threshold can terminate division before the benefits of subsequent splits become evident, while too low a value results in little simplifica-

tion [Qui93].

C4.5 (like CART) follows the second approach. The divide-and-conquer process is given free opportunity and the overfitted tree that it produces is then pruned. Parts of the tree that do not contribute to classification accuracy on unseen cases are removed, producing something less complex and thus more comprehensible. The additional computation invested in building parts of the tree that are subsequently discarded can be substantial, but this cost is offset against benefits due to more thorough exploration of possible partitions. Growing and pruning trees is slower but more reliable [Qui93, EMS93].

Pruning a decision tree will almost invariably cause it to misclassify more of the training cases. Consequently, the leaves of the pruned tree will not necessarily contain training cases from a single class. However, C4.5 enables the user to control the extent of pruning by varying a parameter in the package called the "confidence factor." Small values of this parameter cause more heavily pruning than large values, with the highest effect on domains with smaller sets of cases. In the experimental work of this thesis, the confidence factor is frequently used to control the consistency level (vs. the size) of the learned trees.

2.7.2 Estimating the Generalization Performance

The classical method used by C4.5 to estimate the reliability of a decision tree is to divide the set of cases available into two sets: the training set, which is used to build the decision tree, and the test set used to examine the generalization performance of the tree on the unseen cases. This technique is satisfactory when there is plenty of cases, but often we don't have enough cases to divide. This leads to two problems.

First, in order to get a reasonably accurate error rate, the test set must be large. This will leave the training set represented by few cases which means the generated tree might not reflect the structure of the domain and, therefore, the generalization performance deteriorates. Second, when the total number of cases is not enough (for instance, several hundred cases), different divisions of the set of cases into training and test sets can produce surprisingly large variations in error rates on unseen cases.

However, in order to obtain a more robust estimate of accuracy on unseen cases, C4.5 offers the option to use a well-known strategy called "cross validation." In this strategy, the set of cases is divided into N blocks such that each block's number of cases and class distribution are as uniform as possible. Each time one of the blocks is set aside as a testing set and the remaining $N-1$ blocks are used to build the tree. Next time, a different block is used for tuning and the rest for training and so on. Therefore, each block has only one chance to be used as a testing set. Finally, we

will be having N different decision trees. The average error rate over the N unseen test sets is then expected to be a good predictor of the error rate of the tree built from the set of all cases [Qui93].

2.8 Extensions of ID3

Many researchers attempted to extend the ID3 algorithm in various directions. The basic algorithm assumes no noise in the domain and learns decision trees that classify cases in the training set perfectly; i.e., they are 100% consistent with the training set. However, application to real-world domains require methods for handling noisy as well as inconclusive data and mechanisms that do not overfit the tree [CN89, Qui86]. Tree pruning techniques were used in the ID3 algorithm and proved to be effective against overfitting [EMS93, Sch95]. Furthermore, Quinlan discussed in his paper [Qui86] the modifications required to enable ID3 to operate with a noise-affected training set and presented some possible techniques to achieve this goal.

Inconclusive data is another problem facing learning in real-world domains. In their paper [SFU89], Spangler, Fayyadh, and Uthurusamy described a modified version of ID3, called INFERULE, which addresses some of the problems involved in learning from inconclusive data. An inconclusive data set is a data set of examples

expressed in terms of a set of noise-free attributes which do not contain the necessary information for predicting a unique outcome for each example.

The code of C4.5, which is the latest implementation of the ID3 algorithm, still lacks many extensions such as handling continuous class, ordered discrete attributes and structured attributes.

Continuous classes are needed whenever the class to be predicted, represented as a quantity, is numeric rather than categorical. Some work is done on learning continuous classes in CART [BFOS84].

Another handy extension to ID3 is to have ordered discrete attributes. As for attributes in ID3, they can be either continuous, and hence ordered, or discrete, in which case they are presumed to be unordered. Many systems implement a third kind of attribute, *ordinal*, that has discrete, ordered values. Common uses of ordinal attributes are for qualitative descriptions, such as giving a reading as very low, low, medium, high, or very high. An attribute like this should often be treated similarly to a continuous attribute, with tests that divide the values into those below and those above a threshold value. In C4.5, the only way to represent this kind of attributes is to map the discrete values onto integers (e.g., setting very low=1, low=2, and so on), and then pretend that the attribute is continuous [Qui93].

Furthermore, the ID3 algorithm can not be used in domains with so-called *tree-*

structured attributes, which are discrete attributes having a hierarchy of possible values (an *is-a* hierarchy), rather than just a list of values [AAK95]. As an example, consider an attribute like color that can be specified at different levels of details as shown in Figure 2.10.

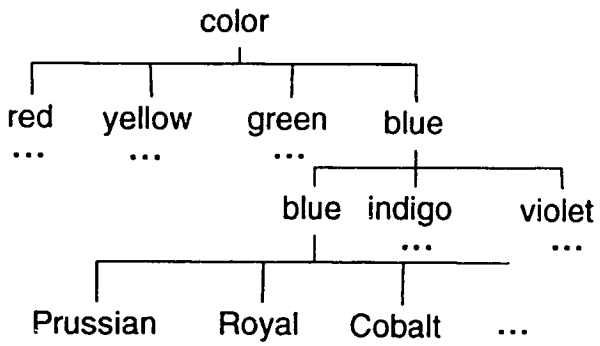


Figure 2.10: The value hierarchy for color attribute.

Quinlan lists the support for this kind of attributes as one "desirable extension" to his C4.5 decision tree learning package [Qui93]. In [AAK95, AAK96], Almuallim, Akiba, and Kaneda described an approach which they developed to handle tree-structured attributes directly compared to other approaches which require preprocessing of the training examples so that they are re-encoded in terms of an equivalent set of purely nominal attributes defined based on the hierarchies of the tree-structured attributes.

Some attempts were made to improve the quality of the tree generated by ID3. For instance, GID3 [Fay91] was developed as a generalized version of ID3 which does

not necessarily branch on each value of the chosen attribute. Instead, it can branch on arbitrary individual values of an attribute and "lump" the rest of the values in a single *default branch.* The default branch represent a subset of the value of the attribute. The idea behind GID3 was to reduce unnecessary subdivision of the data.

Similarly, the GID3* algorithm [Fay91] was developed to improve GID3 so that it can decide which values are relevant enough to be branched on without relying on a user-specified parameter.

Other algorithms were reported that build trees by labelling the test nodes with tests composed of linear combination of the attributes [MKSB93]. Because of the computational intractability of finding an optimal combination of the attributes, heuristic methods were used to produce good trees.

While all the previously mentioned algorithms attempted to extend the ability of the ID3 algorithm to cope with different forms of attributes and classes or reduce unnecessary sub-division of data as in GID3 and GID3* [Fay91], other algorithms try to utilize the tree generated by ID3 in building a new tree and improving over a certain quality such as the average cost or the generalization performance.

Other attempts were made to improve the quality of the current tree using what is called *incremental learning.* There are two situations where we can make use of the current tree in building a new one.

First, in practice, after a tree is generated, new data might become available due to the continual collection of data. It may be that the existing decision tree deals correctly with all the new data, so providing no reason to change the tree under the rule, "if it isn't broken, don't fix it."

However, if the tree does not classify the cases satisfactorily correctly, then we can either ignore the new data or discard the previous tree and build a new one from the union of the new data and the training set. Some algorithms were proposed to modify the current tree in the light of the new data. These algorithms retain information at the test nodes of the tree sufficient to permit its modification. On the other hand, if computation time is available then it is better to discard the current tree and build a new one [Qui93].

The second situation involves being able to spend more time in building a good tree from the same training set. Many algorithms were developed in this direction aiming at improving the quality of the final output tree.

Among these were algorithms which follow the constructive learning approach such as CITRE [MR89] and FRINGE [Pag89]. FRINGE [Pag89] enlarges the set of attributes by introducing new attributes. It builds a decision tree using the initial set of attributes and analyses this tree to find candidates for useful attributes and add them to the set of attributes. Then it re-describes the examples in the training

set (by computing the values of the new attributes for each case) and build a new tree from the set of examples and the new set of attributes. This process is iterated until no new candidates attribute are found [Pag89, PH90].

CITRE [MR89] is similar to FRINGE [Pag89] in its use of learned decision trees to suggest useful constructed attributes. Both systems iterate between tree learning and attribute construction: a decision tree is learned, the tree is used to construct new attributes, the new features are used in building a new tree, and the process is repeated until no new attributes are constructed [MR89].

Our work falls under the class of algorithms aiming at generating better trees in an iterative method. We developed two iterative approaches: the weight adjustment approach and the successive refinement approach. The former approach works by altering the amount with which each case in the training set contribute to the attribute scores in the process of attribute selection. This changes the attributes selected to label test nodes and, hence change the resulting tree. The latter approach is based on our newly developed criterion, called the gain-importance criterion, which is used to improve the choice of attributes in the attribute selection process. Those two approaches will be explained in detail in the chapters 4 & 5.

Chapter 3

Limitations in the Top-Down Approach

The main problem with the top-down approach of interest to us in this thesis is the locality of decision. The decision to select a certain attribute is based solely on the score of the attribute selection criteria of each attribute without having the ability to backtrack and correct a wrong decision or predict that an attribute is not as good or as bad as it seems to be. In other words, a bad choice of an attribute can not be fixed later or predicted earlier.

This problem could result in more splitting which, in turn, leads to more complex trees. This means that, during the construction of the decision tree, the number

of cases used in the attribute selection process at each node will be small which will decrease the confidence in the selected attribute. Another consequence of the locality of decision problem is being unable to capture any relation among attributes which could simplify the tree [Qui93, Fay91].

Consider the boolean expression $(\neg A \wedge \neg B) \vee (A \wedge \neg C)$ where an instance is positive if this formula is *true* and negative otherwise. To illustrate the problem, we created a random data set of 3500 cases in the training set and 3700 cases in the test set using the above formula such that the cases are fabricated in a uniform distribution manner with no bias to the positive nor the negative class. The cases in both the training set and the test set were described by a set of 10 attributes among which the values of attributes A, B, and C were used to determine the class of each case using the above formula and the remaining attributes were irrelevant. The distribution of the attributes in the training set is shown in Table 3.1 and the best decision tree for the above expression is shown in Figure 3.1.

Although this problem may seem only artificial with no real applications in the real-world, this is not the case. Consider the following example.

Let A stands for the occupation of a person, which could be an employee or a student, B stands for the working hours per week, and C stands for the "GPA" of a student. A decision tree for this function is shown in Figure 3.2.

A	B	C	Class(+)	Class(−)
0	0	0	478	0
0	0	1	483	0
0	1	0	0	461
0	1	1	0	418
1	0	0	459	0
1	0	1	0	444
1	1	0	394	0
1	1	1	0	363

Table 3.1: The distribution of the attributes A, B, and C in the training set.

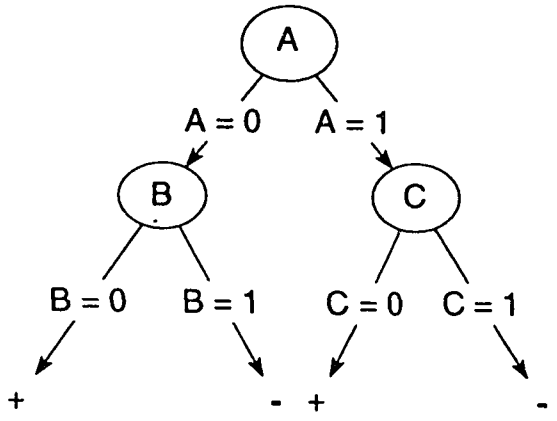


Figure 3.1: The best tree for expression: $(\neg A \wedge \neg B) \vee (A \wedge \neg C)$Avg. Cost = 2.00 tests / case

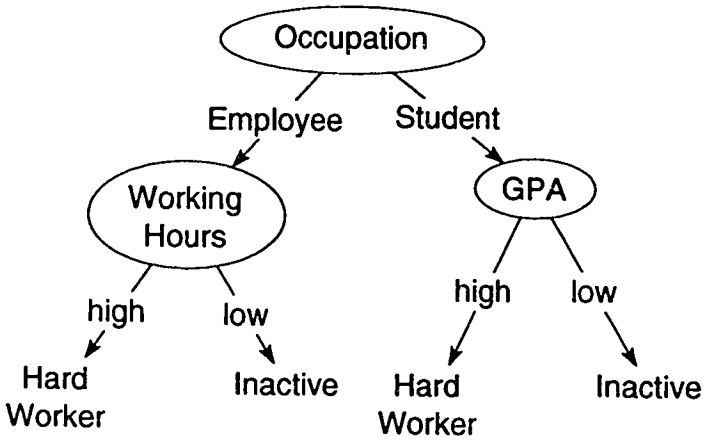


Figure 3.2: The best tree for people classification domain.

When we ran C4.5 on our set of random cases to build a decision tree, C4.5 generated the tree shown in Figure 3.3. As this figure shows, C4.5 chose B instead of A for splitting at the root because B scored higher in the information-gain measure.

The scores were computed as follows:

$$info(S) = -\frac{1814}{3500} \times \log_2(\frac{1814}{3500}) - \frac{1686}{3500} \times \log_2(\frac{1686}{3500}) = 0.999 \text{ bits.}$$

$$\text{For } A: \ info_X(S) = \frac{1840}{3500} \times (-\frac{961}{1840} \log_2 \frac{961}{1840} - \frac{879}{1840} \log_2 \frac{879}{1840}) + \frac{1660}{3500} \times$$
$$(-\frac{853}{1660} \log_2 \frac{853}{1660} - \frac{807}{1660} \log_2 \frac{807}{1660}) = 0.999 \text{ bits..}$$

The information-gain is $0.999 - 0.999 = 0.000$ bits. C4.5 will immediately decide that A is a bad selection since the information-gain measure scored very low.

$$\text{For } B: \ info_X(S) = \frac{1864}{3500} \times (-\frac{1420}{1864} \log_2 \frac{1420}{1864} - \frac{444}{1864} \log_2 \frac{444}{1864}) + \frac{1636}{3500} \times$$

$$(-\tfrac{394}{1636}\log_2\tfrac{394}{1636} - \tfrac{1242}{1636}\log_2\tfrac{1242}{1636}) = 0.794 \text{ bits}..$$

The information-gain is $0.999 - 0.794 = 0.205$ bits. B is a possible candidate for selection.

For C : $info_X(S) = \tfrac{1792}{3500} \times (-\tfrac{1331}{1792}\log_2\tfrac{1331}{1792} - \tfrac{461}{1792}\log_2\tfrac{461}{1792}) + \tfrac{1708}{3500} \times$
$(-\tfrac{483}{1708}\log_2\tfrac{483}{1708} - \tfrac{1225}{1708}\log_2\tfrac{1225}{1708}) = 0.841$ bits.

The information-gain is $0.999 - 0.841 = 0.158$ bits. C4.5 will calculate the information-gain for the remaining irrelevant attributes which are expected to score very low. Finally, it will pick B because the information-gain is maximized among the attributes A, B, and C even though A is a better choice and will generate a more compact tree.

This will happen no matter how large the number of cases is (as long as they are drawn according to the uniform distribution). A will always get a very bad score under the information-gain measure even though it is the best attribute to use. The importance of A is detected one level later in this case after the less important attribute B has already been selected at the root of the tree. This undesirable selection occurs because the decision is made locally without being able to predict that A is a better choice than B or to correct wrong decisions by means of backtracking.

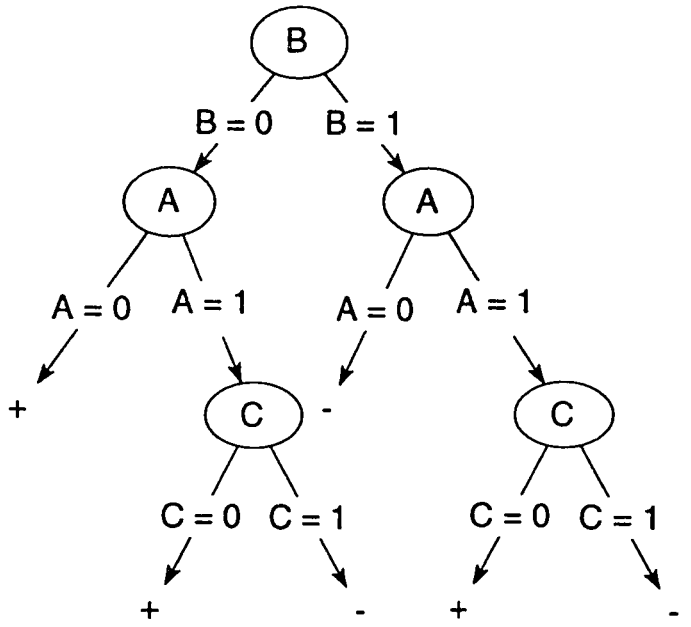


Figure 3.3: The decision tree generated by C4.5 for expression: $(\neg A \wedge \neg B) \vee (A \wedge \neg C)$
Avg. Cost = 2.47 test / case

Consider the decision tree in Figure 3.4 which generated shows the decision tree generated by C4.5 for the people classification problem. With respect to the average cost, there will be an incurred cost for every student due to the test on "Working Hours" attribute. This test is, in fact, redundant for students since they are classified based on their GPA score not how many hours they work. Therefore, there is an additional cost of one extra test for every student case.

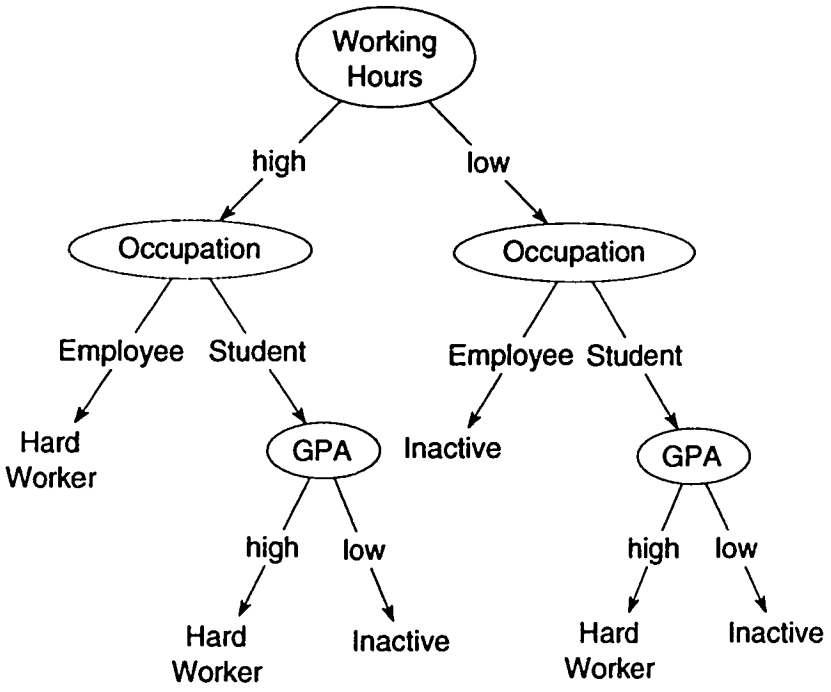


Figure 3.4: The decision tree generated by C4.5 for people classification domain.

On the other hand, due to splitting on the "Working Hours" attribute, more cases might go into one branch than the other based on the class distribution of employees. In other words, if most employees have high working hours, then less cases will go into the branch with low working hours. This, in turn, will lead to even less cases going into the branch labelled student. At this point, it is possible that having small number of cases might favour a different attribute to be selected other than "GPA", or worse, C4.5 might decide to stop splitting because no attribute scores good with respect to the attribute selection criterion. Consequently, the tree generalization performance decreases.

One way to fix this locality of decision problem is to use either a lookahead approach or to backtrack and correct a wrong decision made earlier. However, both approaches are expensive solutions.

In this thesis, we keep the locality of decision and try to solve the problem by using iterative approaches. We generate the first tree, then we try to improve it and generate a second tree. Then, we try to generate a better tree than the second one and so on.

Chapter 4

Improving Decision Tree Learning by Examples' Weights Adjustment

4.1 The Idea of Weighted Cases

In the existing decision tree learning algorithms, each case carries the same weight. Before choosing an attribute at the current test node, some score, such as the information-gain or the gain-ratio, is computed for each attribute. During this computation, each case contributes an amount of exactly 1 in the counts. For example,

when computing the entropy

$$-\sum_{i=1}^{k} \frac{freq(C_i,S)}{|S|} \log_2 \left(\frac{freq(C_i,S)}{|S|} \right),$$

each case contributes the amount of 1 in $|\ S\ |$ and $freq(C_j, S)$. In this work, we propose assigning a positive real-valued weight to each case. Let us denote the weight of a case e by $w(e)$. Under this setting, case e will contribute the amount of $w(e)$, rather than exactly 1, in any counting. Thus, the entropy will be computed as

$$-\sum_{i=1}^{k} \frac{\sum_{e \in S_i} w(e)}{\sum_{e \in S} w(e)} \log_2 \left(\frac{\sum_{e \in S_i} w(e)}{\sum_{e \in S} w(e)} \right)$$

where S_i stands for the set of cases belonging to class C_i.

To illustrate the effect of assigning weights to cases, let us go back to the problem of learning the function $(\neg A \wedge \neg B) \vee (\ A \wedge \neg C)$ discussed in Chapter 3. In that problem, attribute A scored very bad under the information-gain measure, although it is the best choice at the root of the tree. Altering the weights of the cases will obviously change the scores of the attributes, and hence, change the resulting tree.

The interesting question is: "Can the weights of the examples be changed such that attribute A get selected instead of attribute B in the above problem?" The answer to this question is yes as shown in the following computations.

After generating the first tree using C4.5 with the weight of each case set to 1, suppose we increase the weights of the cases as shown in Figure 4.1. The new scores for attributes A, B, and C will then be as follows:

$$info(S) = -\frac{1870}{3500} \times \log_2(\frac{1870}{3500}) - \frac{1630}{3500} \times \log_2(\frac{1630}{3500}) = 0.997 \text{ bits.}$$

$$\text{For } A: \ info_X(S) = \frac{1880}{3500} \times (-\frac{1530}{1880}\log_2\frac{1530}{1880} - \frac{350}{1880}\log_2\frac{350}{1880}) + \frac{1620}{3500} \times (-\frac{340}{1620}\log_2\frac{340}{1620} - \frac{1280}{1620}\log_2\frac{1280}{1620}) = 0.716 \text{ bits.}$$

The information-gain is 0.997 – 0.716 = 0.281 bits.

$$\text{For } B: \ info_X(S) = \frac{2532}{3500} \times (-\frac{1771}{2532}\log_2\frac{1771}{2532} - \frac{761}{2532}\log_2\frac{761}{2532}) + \frac{968}{3500} \times (-\frac{204}{968}\log_2\frac{204}{968} - \frac{764}{968}\log_2\frac{764}{968}) = 0.844 \text{ bits.}$$

The information-gain is 0.997 – 0.844 = 0.153 bits.

$$\text{For } C: \ info_X(S) = \frac{1280}{3500} \times (-\frac{1100}{1280}\log_2\frac{1100}{1280} - \frac{180}{1280}\log_2\frac{180}{1280}) + \frac{2220}{3500} \times (-\frac{768}{2220}\log_2\frac{768}{2220} - \frac{1452}{2220}\log_2\frac{1452}{2220}) = 0.804 \text{ bits.}$$

The information-gain is 0.997 – 0.804 = 0.193 bits.

As a result, attribute A is selected since it has the best score among the three attributes.

The above example shows that assigning weights to cases alters the scores of the attributes, and hence, changes the resulting decision tree.

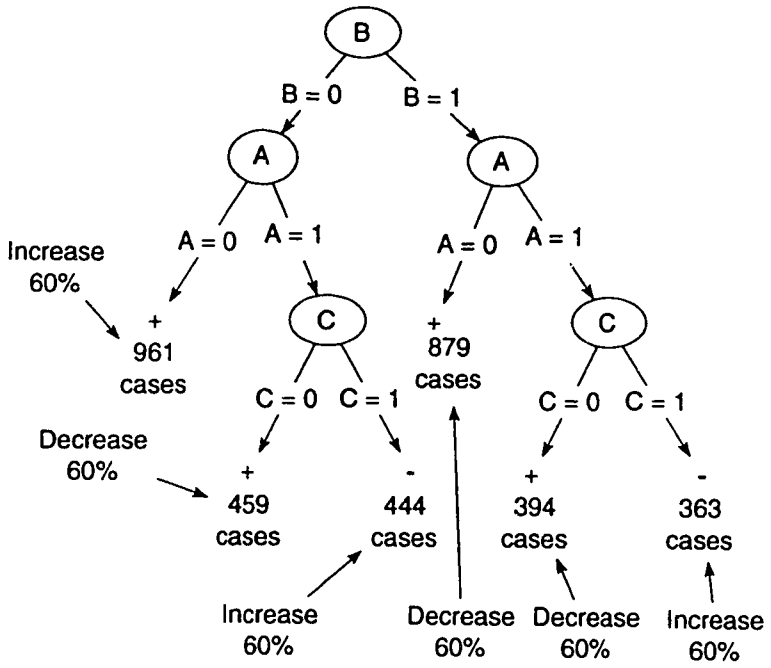


Figure 4.1: Modifying the cases' weights so that A is favoured.

The modified version of C4.5 which considers the weights of the cases will be referred to as C4.5W. Note that C4.5W is deterministic. That is, running C4.5W on a set of cases with fixed weights will always result in a unique tree because the attribute scores will always be the same. Our goal then becomes to determine the weights that lead to a decision tree of improved quality. The following section gives a helpful view for achieving this goal.

4.2 The Space of Weight Vectors

Assuming some arbitrary ordering on the set of cases, the weights of these cases can be viewed as a vector of weights, $W =< w_1, w_2, w_3, \cdots, w_n >$, where w_i is the weight of the ith case in S and n is the total number of cases in S. Due to the determinism property of C4.5W, every weight vector determines a unique decision tree; that is, the tree constructed by C4.5W. Figure 4.2 illustrates this relation between the space of decision trees consistent with a given set of cases and the space of weight vectors for that set. Each point in the latter space corresponds to a vector of weights that generates a unique tree in the space of decision trees.

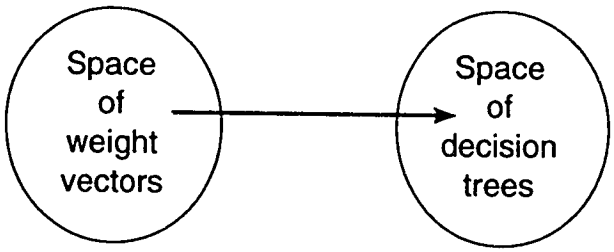


Figure 4.2: The Relation between the space of weight vectors for a given set of cases and the space of decision trees consistent with that set.

With this view of cases' weights as a vector, searching for a better tree in the space of decision trees means looking for the weights vector which produces this tree.

Our proposed approach adopts this view because it is more convenient to use.

Adjusting the cases' weights becomes a matter of adding an update vector to the old vector of weights. The initial vector W_0 is set as $W_0 =< 1, 1, 1, \cdots, 1 >$. This vector is used to build the first tree which is equivalent to the one generated by C4.5. Then, an update vector is computed and added to the current vector of weights to form a new vector which is used to build a new decision tree. This process is repeated until no more improvement in tree quality is possible.

Formally, the new vector of weights is computed as follows: $W_{new} = \alpha + W_{old}$. The update vector α is of the form $\alpha =< \alpha_1, \alpha_2, \alpha_3, \cdots, \alpha_n >$ where α_i is a real-valued factor, which could be positive or negative, and it is to be added to w_i in W_{old}. Since each weight w_i in W_{new} should have a positive real-value, each α_i in α vector should be chosen carefully. In this work, we allow only small values for α_i's in order to avoid building trees with great variations in their quality and to gradually reach the vector of weights which yields the best tree.

Basically, two methods are used in this thesis to compute the update vector α: random and careful updating. In the random updating method, α_i's are computed at random. The second method, careful updating, computes the new weight of any case e based on the difference between the depth of the leaf which classifies it in the last generated tree T_{old} and the average depth of that tree. The idea is to increase the weights of the cases classified by deeper leaves in the tree and, conversely, decrease the weights of those classified by leaves in the upper parts of the tree. More details

about both approaches will be given in the next section.

4.3 Simulated Annealing Strategy

Having converted the search in the space of decision trees into a search in the space of weights vectors, we need to find an efficient strategy to handle the search in the huge space of vectors. Obviously, an exhaustive search through this space for the vector which produces the optimal tree is impractical. The alternative would be to use one of the iterative improvement schemes where each iteration brings us one step closer to the optimal tree [LHS95].

The idea behind iterative improvement schemes is as follows. If the new vector of weights W_i, computed from α and W_{i-1}, leads to a better decision tree T_i, the tree is accepted and a new update vector of weights α should be formulated. The next vector W_{i+1} is then computed from W_i and the new update vector α and a new tree T_{i+1} is built. However, if the new vector of weights W_i leads to a worse tree, then a different update vector of weight α should be formulated and used with W_{i-1} to compute a new vector W_i. This new vector is then used to generate a new tree and this tree is tested as before.

Hill climbing is a well known iterative improvement scheme. It proposes accept-

ing a vector W_{i+1} only if it leads to a better decision tree. Otherwise, a different update vector α is formulated and W_{i+1} is re-computed from W_i and the new update vector α.

The major drawback of hill climbing is the possibility of getting stuck at local maxima where all update vectors α lead to worse trees. *Simulated annealing* is often described as a strategy to escape getting stuck at local maxima by accepting update vectors leading to worse trees with a certain probability.

Simulated annealing has been a useful local search technique for obtaining near-optimal solutions in a wide variety of combinatorial optimization problems. It is based on ideas in statistical mechanics, which is concerned with the behavior of systems with many degrees of freedom in thermal equilibrium at a finite temperature. However, combinatorial optimization is concerned with finding the minimum of a given function depending on many parameters [LHS95, KGJV83].

To understand the relation between simulated annealing and statistical mechanics of annealing in solids such as crystal lattice, we consider how to transform a solid into a low energy state. A low energy state usually means a highly ordered state, such as a crystal lattice. A relevant example here is the need to grow silicon in the form of highly ordered, defect free crystals for use in semiconductor manufacturing. To accomplish this, the material is *annealed*: heated to a temperature that

permits many atomic rearrangements, then cooled carefully, slowly, until the material freezes into a good crystal. Simulated annealing uses a similar set of controlled cooling operations for nonphysical optimization problems [Rut89].

In practice, simulated annealing has proven to be a very powerful tool for many of the large scale technological problems facing engineers and scientists today. It has been applied to large scale combinatorial optimization problems such as VLSI design, pattern recognition, and graph partitioning with excellent results [PD95].

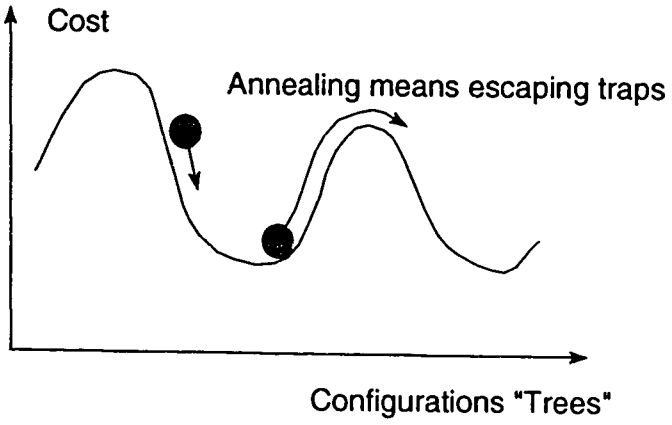


Figure 4.3: Simulated Annealing.

The concept behind simulated annealing is to accept any vector W_{i+1} if it leads to a better decision tree, as in hill climbing. However, simulated annealing differs from hill climbing in dealing with weight vectors leading to worse trees. Such vectors are accepted with a certain probability p. This probability p is inversely related to the *difference in quality*, Δ, between the tree generated using W_i and the current tree

generated using W_{i+1}. Moreover, this probability has to be decreased as we proceed in our search. For this purpose, simulated annealing uses a parameter called $Temp$ whose value is reduced slowly through the iterations. The probability p of accepting a worse solution is computed as follows.

$$p = \frac{1}{1+e^{\frac{-\Delta}{Temp}}}$$

where Δ = Current tree quality − Last tree quality. At the beginning, we start with a high temperature which enables more weight vectors leading to worse trees to be accepted. Then, according to a certain cooling schedule, this temperature is lowered to reduce the probability of acceptance for those vectors.

4.4 An Outline of the Approach

The approach proposed in this chapter for improving decision trees quality is outlined as follows:

1. Let $W_0 =< 1, 1, 1, \cdots, 1 >$. That is, initialize the weight of each case in the training set to 1.

2. Run C4.5W (a modified version of C4.5 which considers the cases' weights) to form the initial tree T_0 and set this tree as the last accepted tree T_{last} as well

as the best accepted tree so far T_{best}.

3. Initialize the cooling schedule: the starting temperature, rules to determine when to lower the temperature, by how much should it be lowered, and when to stop annealing.

4. Formulate the update vector of weights α and compute the new vector of weights W_{i+1} as $W_i + \alpha$, where $i = 0$ initially.

5. Run C4.5W to form a new tree $T_{current}$ using W_{i+1}.

6. If $T_{current}$ is not satisfactorily consistent with the training set (due to pre-pruning) then go to step 12.

7. Calculate the quality score of $T_{current}$ with respect to the evaluation criterion. (We will use the average cost and the generalization performance of the tree as the evaluation criteria in this work)

8. If $T_{current}$ is better than T_{last} then accept it as T_{last} (and if it is better than T_{best} then accept it as T_{best}), increment i, and go to step 12.

9. Otherwise, compute the probability p of acceptance of $T_{current}$ as a function of the current temperature as well as the difference between the quality of $T_{current}$ and T_{last}.

10. Accept $T_{current}$ as T_{last} with probability p. This is done by generating a random number between 0 and 1 then comparing it with p. If it is less than p then

accept $T_{current}$ as T_{last} and increment i. Otherwise, reject it.

11. Reduce the temperature according to the cooling schedule.

12. Formulate a new update vector of weights α and compute a new vector of weights W_{i+1}.

13. If the pre-specified number of iterations is exceeded then return T_{best}. Otherwise, go to step 5.

As we mentioned previously, two methods could be used to formulate the update vector of weights: random updating and careful updating. In both methods, the new weight of each case $w(e)_{new}$ is computed as follows.

$$w(e)_{new} = (1+ \textit{ update factor }) \times w(e)_{old}$$

The value of *update factor* depends on what method of updating is used, as we will see later in this section. Note that the sum of the weights of the cases in tree T_0 is equal to the number of cases in the training set because each case has a weight of 1. But, when a factor is added or subtracted from the old weight, the sum of weights may become larger or smaller than the number of cases. Therefore, the new vector of weights has to be normalized in order to keep the effect of weight updates on the resulting tree small and escape having greatly different trees. This

normalization process will keep the sum of the weights equal to the number of cases in the training set.

The idea behind careful updating is to give tests at the lower part of the last generated tree a chance to be selected earlier. The value of *update factor* is a function of the difference between the average depth of the last accepted tree and the depth of the leaf which classifies the case. This difference is multiplied by a small random factor between 0 and 0.01 to give a randomness property to the new weight. The need for this randomness property is explained in the next paragraph. The *update factor* is calculated as follows.

$$update\ factor = (depth(leaf) - average\ depth(T_i)) \times random\ factor$$

The randomness property is needed when the update vector of weights needs to be re-formulated. The new update vector should be different from the current vector which lead to an un-acceptable decision tree. In the case of careful updating, without this randomness property the new vector of weights would be exactly the same as the current one leading to the same un-acceptable decision tree. In fact, the *random factor* acts as a tool to increase or decrease the effect of the difference between the average depth of the current decision tree and the depth of the leaf which classifies each case.

The other method used to formulate update vectors is the random updating method. With this method, the value of each α_i in the *update factor* chosen randomly between -0.1 and 0.1.

4.5 Weight Adjustment to Lower the Average Cost

As mentioned previously, the average cost is one of the important criteria used in practice to evaluate decision trees. It is defined as the average number of tests that must be performed and satisfied before determining the class to which a case belongs. Less average cost value means less number of tests are performed in order to classify a case on average, and, in turn, classification cost becomes more economical.

The goal in this section is to evaluate our approach which employs simulated annealing to generate more economical decision trees.

4.5.1 Experimental Results 1

To achieve the above goal, we performed an experiment on several artificial and natural databases available from the UCI Machine Learning Repository. For more

information about the origins, past usage, and other details of these domains, refer to appendix A.

The experiment was run with pre-pruning enabled. Pre-pruning involves assessing the splits from the point of view of the information gain. If the assessment falls beyond a threshold, splitting is stopped and the tree or the subtree at the current subset of cases is replaced by a leaf. Pre-pruning is discussed in Chapter 2.

Two methods were used to update weight vectors: random updating and careful updating. Both methods were described in details in the previous section. In addition, post-pruning is also used to simplify the resulting tree and lower the effect of overfitting of the data. Post-pruning was also discussed in Chapter 2.

The goal of this experiment is to compare the average cost of the tree generated by C4.5 with that of the tree generated by our approach. For fairness, only trees of similar degrees of consistency with the training set were compared. We control the consistency degree of any tree with the training set by increasing or decreasing the level of pruning, as discussed in Chapter 2. More pruning generates a smaller but less consistent tree. We generate two trees one of which using C4.5 and the other using our approach. Then, we take the tree which is more consistent with the training set, alter the level of pruning and re-generate it. We repeat this process of altering the level of pruning and re-generating until we get a similar degree of

consistency with the training set as the other tree.

We have two sets of experimental results corresponding to our two methods of weights updating: the careful and the random updating methods.

Careful Updating of Weights

Figure 4.4 shows the results comparing the average cost of the trees generated by C4.5 to those generated by our approach with careful updating of weights.

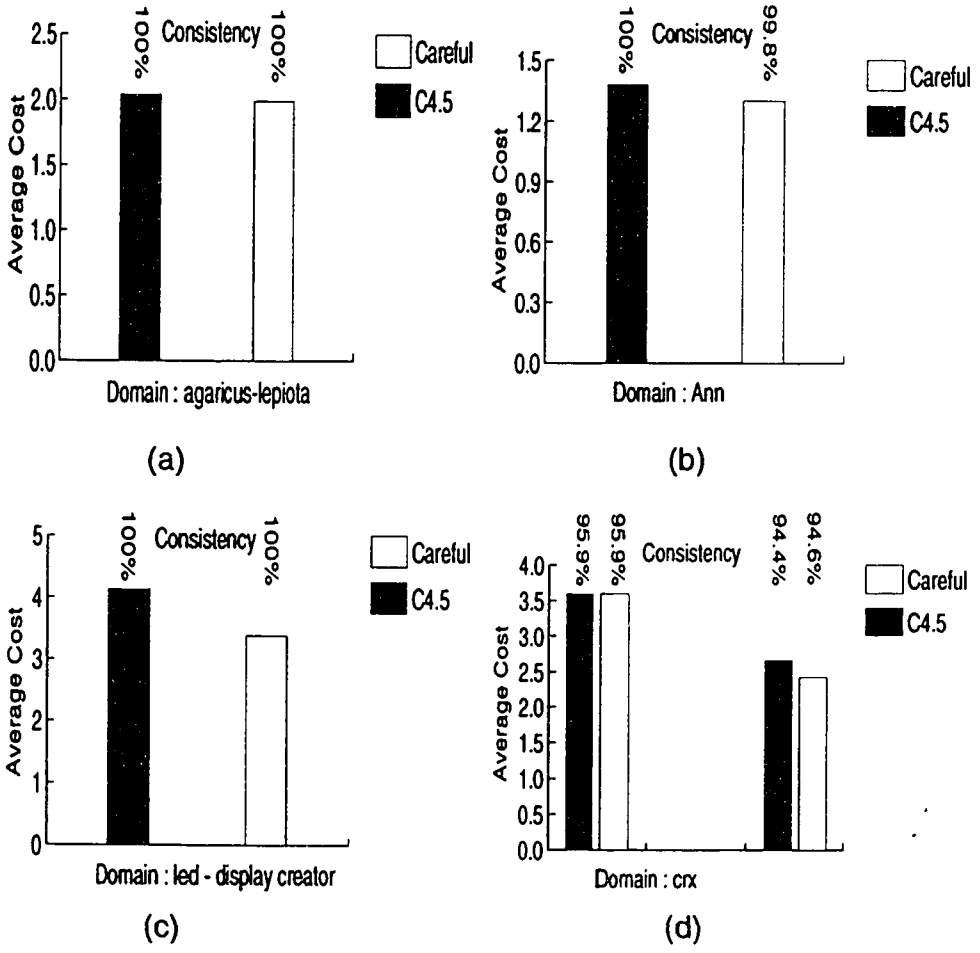
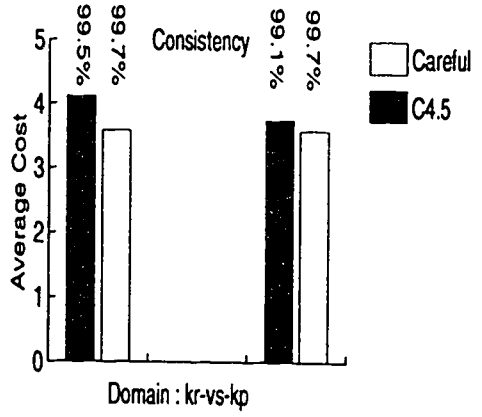
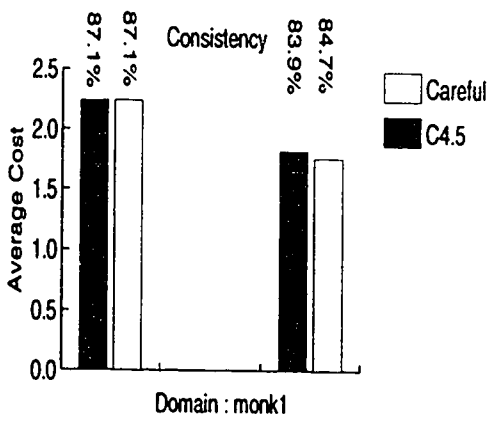


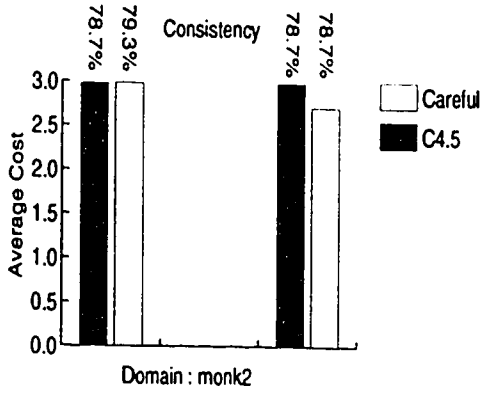
Figure 4.4: A comparison between C4.5 and the weight adjustment approach (using careful updating) in terms of the average classification cost.


99.5%
99.7%
Consistency
99.1%
99.7%
Careful
C4.5
Average Cost
Domain : kr-vs-kp

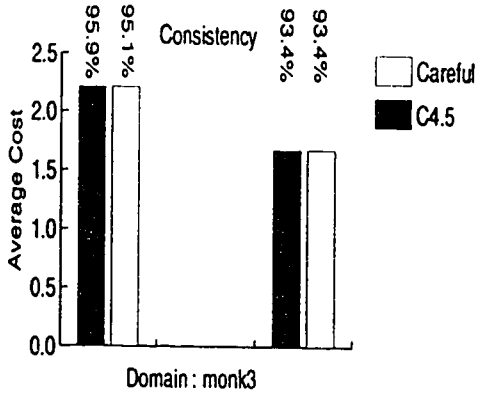
(e)


87.1%
87.1%
Consistency
83.9%
84.7%
Careful
C4.5
Average Cost
Domain : monk1

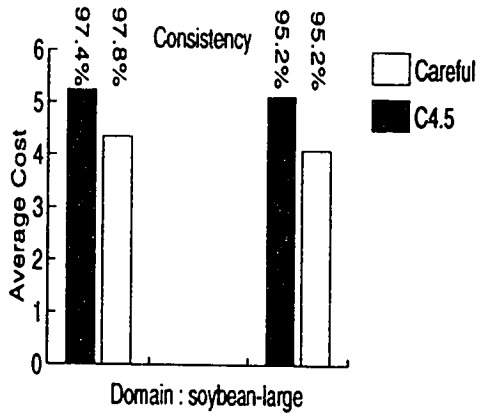
(f)


78.7%
79.3%
Consistency
78.7%
78.7%
Careful
C4.5
Average Cost
Domain : monk2

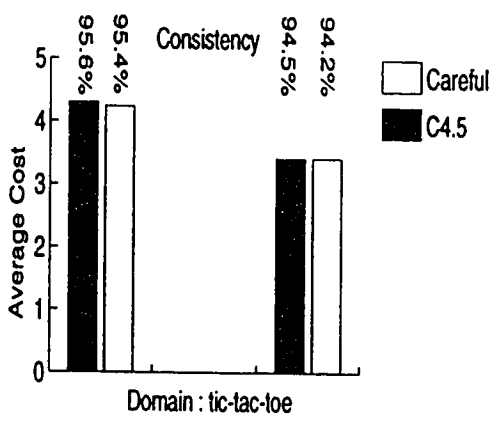
(g)


95.9%
95.1%
Consistency
93.4%
93.4%
Careful
C4.5
Average Cost
Domain : monk3

(h)



(i)



(j)

By comparing trees generated using our approach and those generated using C4.5, we see that our approach leads to trees with reduced average cost in the domains: agaricus-lepiota, led-display-creator, crx, kr-vs-kp, monk1, and soybean-large. Thus, these results suggest that obtaining more economical decision trees than those generated by C4.5.

Random Updating of Weights

Figure 4.5 shows the results we got from running our experiment on various domains adopting the method of random updating of weight vectors.

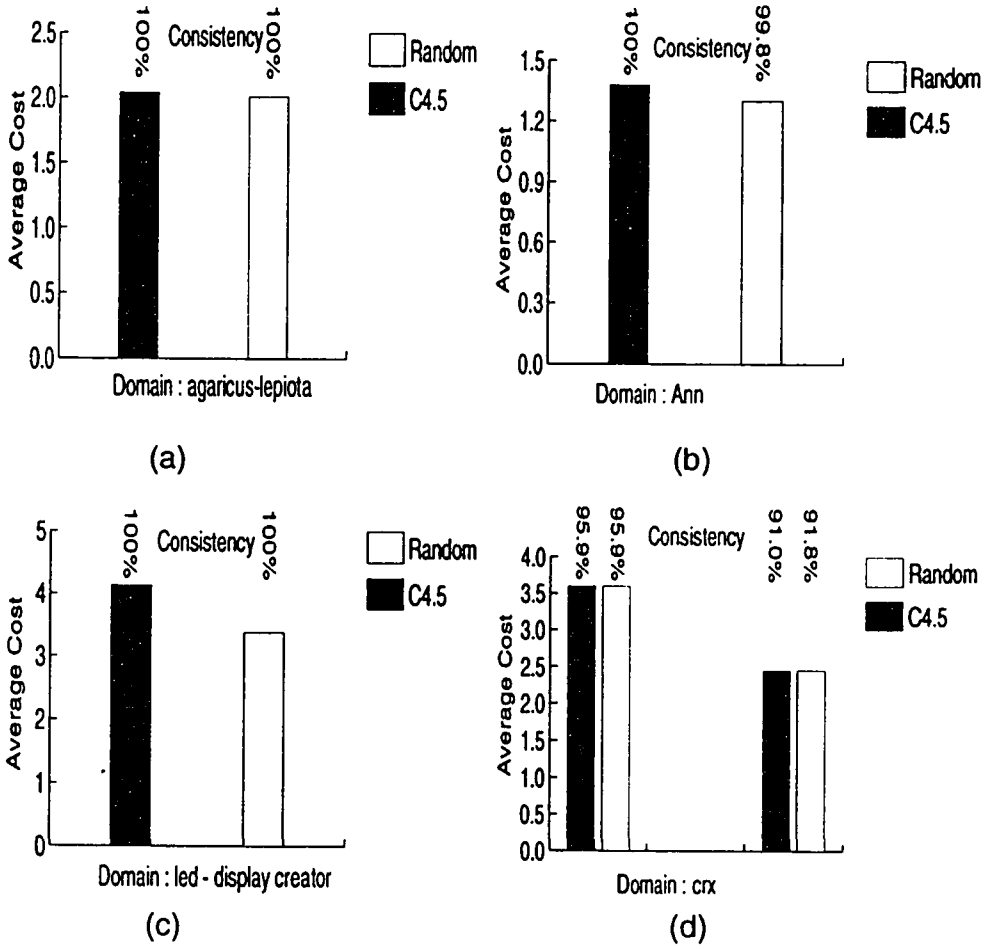
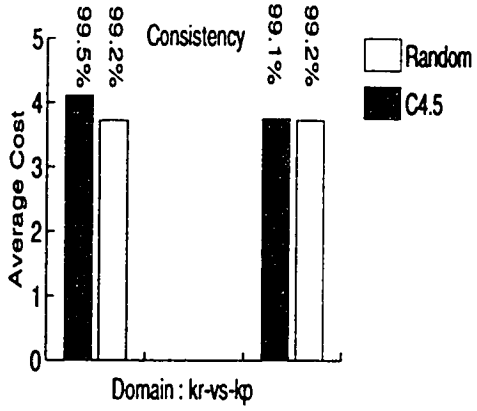
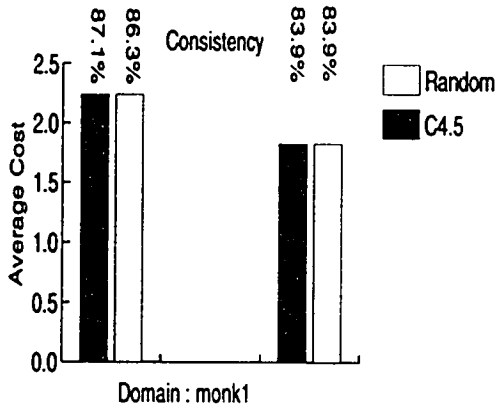


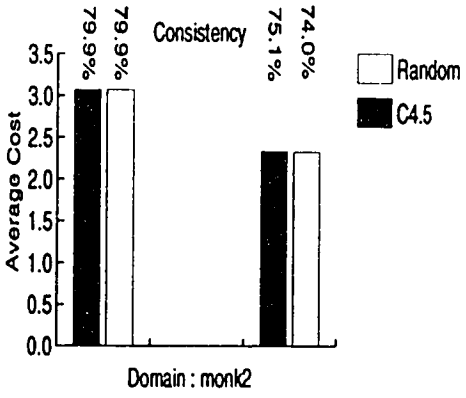
Figure 4.5: A comparison between C4.5 and the weight adjustment approach (using random updating) in terms of the average classification cost.


Consistency
99.5%
98.2%
99.1%
99.2%
Random
C4.5
Average Cost
Domain : kr-vs-kp

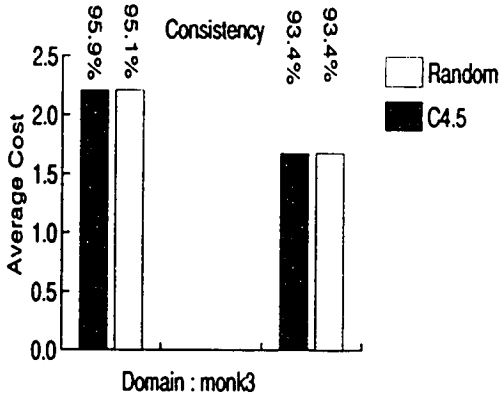
(e)


Consistency
87.1%
86.3%
83.9%
83.9%
Random
C4.5
Average Cost
Domain : monk1

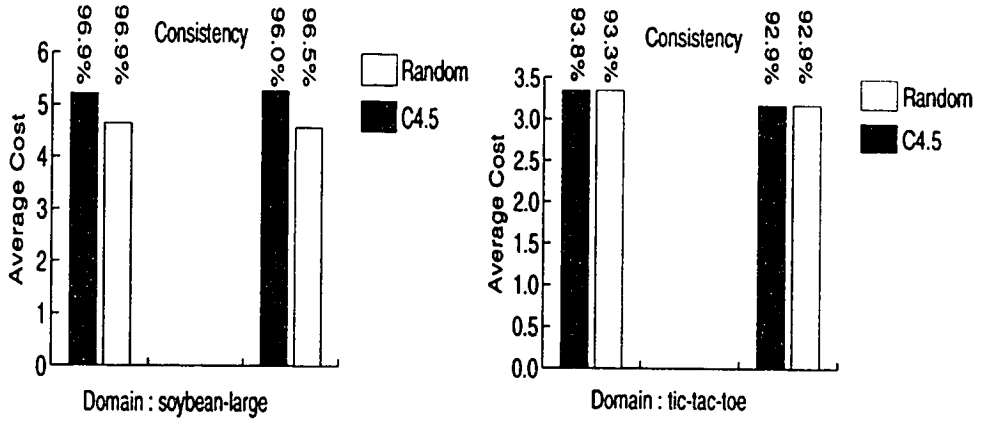
(f)


Consistency
76.9%
76.9%
75.1%
74.0%
Random
C4.5
Average Cost
Domain : monk2

(g)


Consistency
95.9%
95.1%
93.4%
93.4%
Random
C4.5
Average Cost
Domain : monk3

(h)



By comparing trees with similar degrees of consistency, our method wins in the following domains: agaricus-lepiota, led-display-creator, kr-vs-kp, and soybean-large. The number of domains where we were able to obtain more economical trees than those generated by C4.5 decreased compared to the careful updating mode. This is not surprising since we are updating the weight vectors totally at random. As the chance of landing on a good tree increases, the chance of landing on a worse tree also increases. However, our approach was still able to achieve good results in four domains.

4.6 Weight Adjustment to Improve the Generalization Performance

In this section, the goal is to generate trees with better generalization performance. Note that, using the accuracy over the test set to accept or reject the trees generated through the simulated annealing strategy is inappropriate, since C4.5 is never given the chance to access the test set. For this reason, we follow a cross validation strategy using the training examples alone during simulated annealing.

4.6.1 Cross Validation Strategy

Cross validation is a well known strategy used to obtain a more robust estimate of accuracy on unseen cases. We employ this strategy here, however, to estimate the weight vector that leads to the best generalization performance. The strategy works as follows. The test set is set aside and the cases in the training set are splitted into N blocks such that class distribution of cases among the blocks is as uniform as possible. Each time one of the blocks is used as a *tuning* set and the remaining $N-1$ blocks are used to build a tree using the simulated annealing strategy to improve the generalization performance over the tuning set. The vector of weights used to build the tree with the best generalization performance is, then, saved to be used

later.

Next time a different block is used as a tuning set and the remaining $N-1$ blocks are used as a training set. Again, simulated annealing is used to build the tree with the best generalization performance over this tuning set. This process of using a different block as a tuning set and the remaining $N-1$ blocks as the training set goes on N times so that each block is used as a tuning set exactly once.

The above process will eventually give N decision trees with N vectors of weights, each of which was used to build one of those trees. Then, the k most accurate trees among these N trees are selected and their vectors of weights are averaged. At this point, each case in the original training set (the union of the N blocks) will have an average weight computed from the best k vectors of weights.

The final tree is constructed from the original training set (the union of the N blocks) using the average vector of weights. The generalization performance of the final tree is then estimated using the original test set, which was set aside at the beginning, and compared with that of the tree generated by C4.5.

Note that the number of blocks, N, is crucial to the success of the cross validation strategy. A small number of divisions means a small portion of the total cases will go into the training set each time and the resulting tree will be less consistent with the original training set and the accuracy of the tree is expected to be very low over

the un-seen cases in the original test set.

On the other hand, a large number of divisions means less number of cases will go in the tuning set and the resulting tree is expected to have the same accuracy as applying C4.5 over the un-seen cases since the tree will be similar to that generated by C4.5 on the original training set involving all divisions. Both choices have negative effects on the resulting tree. A common choice of N is 10 [Qui93].

4.6.2 An Outline of the Experiment

The experiment of this section was conducted as follows:

1. Shuffle the cases in the training set and split them into 10 near-equal blocks at random. The class distribution of the cases in each block should be as uniform as possible.

2. Choose 9 blocks to form the training set and set the remaining block as the tuning set for the current iteration.

3. Run C4.5W (the modified version of C4.5 which considers the cases' weights) to form the initial tree T_0 and set it as T_{best}.

4. Formulate the update vector of weights α using the random updating method and compute the new vector of weights W_{i+1} as $W_i + \alpha$, where $i = 0$ initially.

5. Run C4.5W to form a new tree $T_{current}$ using W_{i+1}.

6. Compute the generalization performance score of $T_{current}$ with respect to the tuning set.

7. If $T_{current}$ is better than T_{last} then accept it as T_{last} (and if it is better than T_{best} then accept it as T_{best}) and go to step 11.

8. Otherwise, compute the probability p of acceptance of $T_{current}$ as a function of the current temperature as well as the difference between the generalization performance of $T_{current}$ and T_{last}.

9. With probability P, accept $T_{current}$ as T_{last}. This is done by generating a random number between 0 and 1 then comparing it with p. If it is less than p then accept $T_{current}$. Otherwise, reject it.

10. Decrease the temperature based on the cooling schedule.

11. Formulate a new update vector of weights α and compute a new vector of weights W_{i+1}.

12. If the pre-specified number of iterations is exceeded then go to step 13. Otherwise, go to step 5.

13. Save the vector of weights which lead to the tree T_{best} with the best generalization performance over the tuning set.

14. If every blocks has already had only one chance to be as a tuning set, then go to step 15. Otherwise go to step 2.

15. Choose the best 5 trees among the 10 best trees and average their vectors of weights.

16. Use the average vector of weights to generate the final tree from the original training set (the union of all the 10 blocks).

17. Compare the generalization performance of the final tree on the test set (which was set aside at the beginning) to that of the tree generated by C4.5.

4.6.3 Experimental Results 2

The experiment we have just outlined was carried out to evaluate our approach which is based on the simulated annealing strategy combined with cross validation. The experiment was run on several artificial and natural databases available from the UCI Machine Learning Repository. For more information about the origins, past usage, and other details of these domains, refer to appendix A.

The approach was run with pre-pruning enabled. Pre-pruning was discussed previously in Chapter 2. The random updating method was used to update the vectors of weights during the process of searching for the vector of weight which

leads to a better tree. This method was described earlier in this chapter.

Figures 4.6 shows the results of the experiment which compares the generalization performance of the tree generated by C4.5 to the tree generated by our approach.

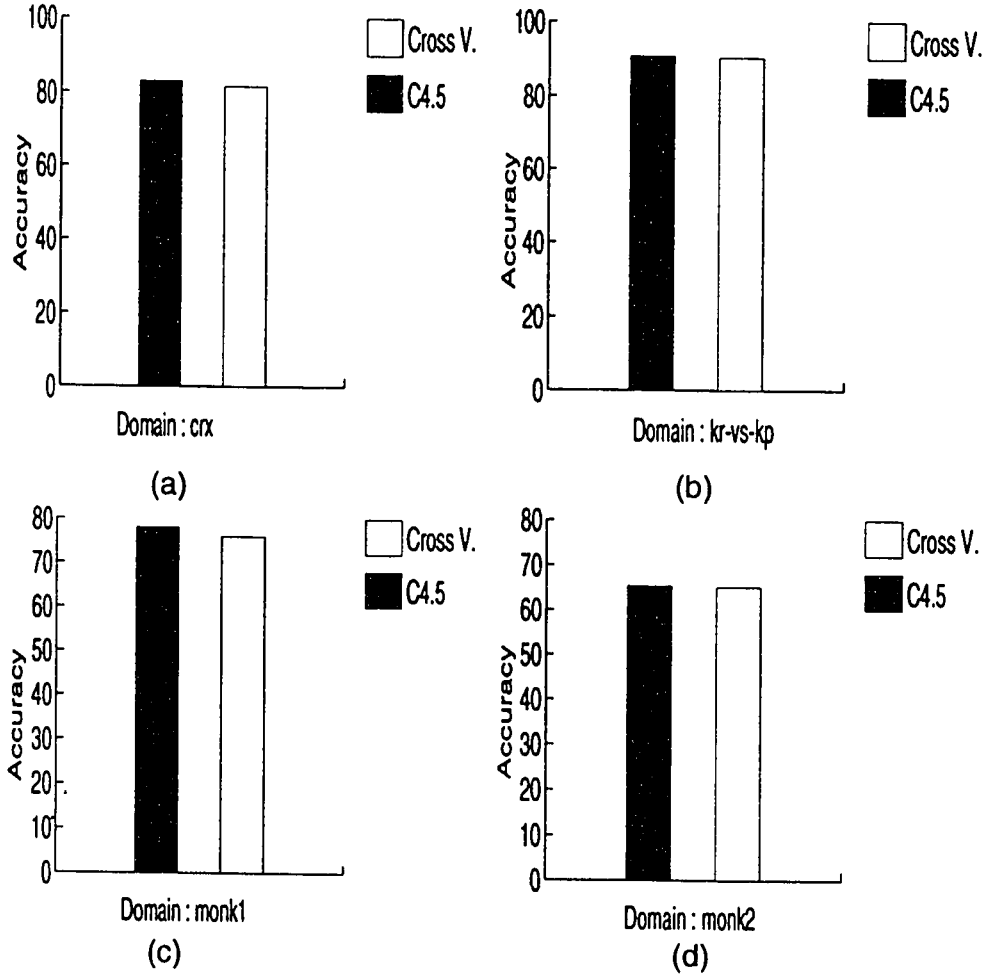
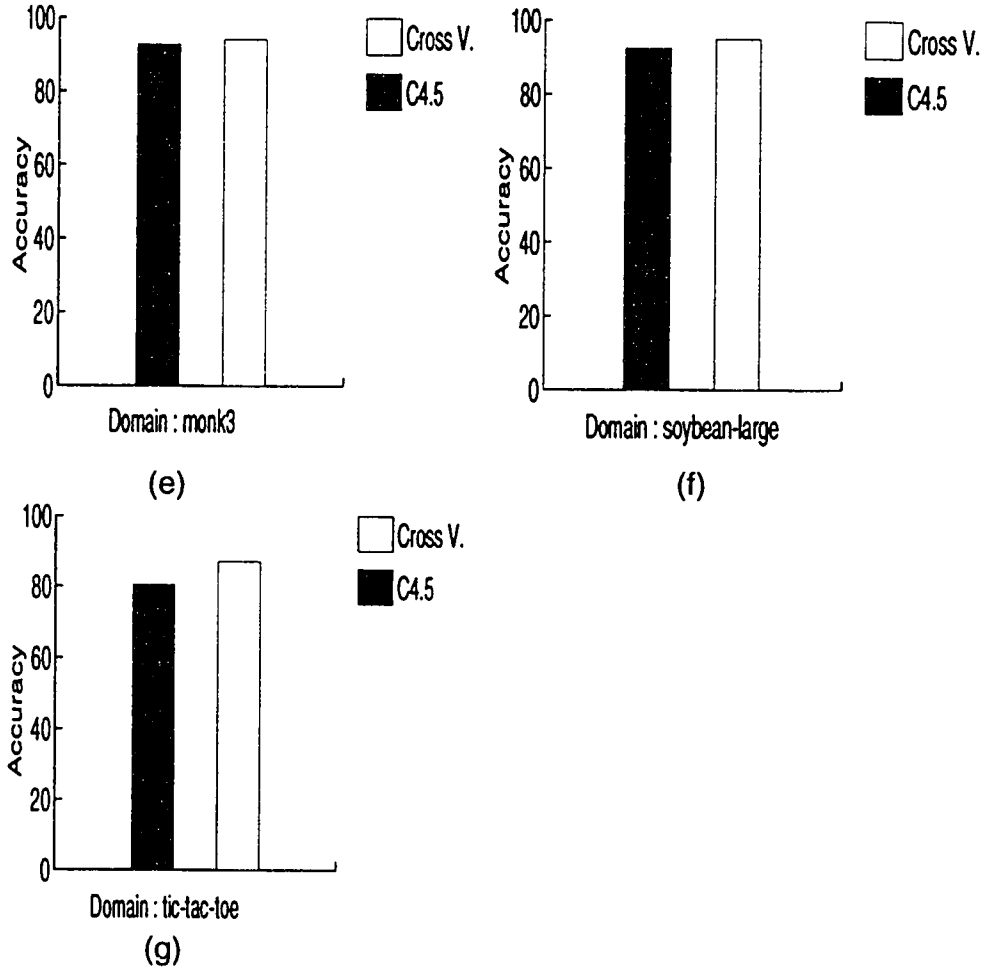


Figure 4.6: A comparison between C4.5 and the weight adjustment approach in terms of the generalization performance of the tree.



By comparing the generalization performance of the trees generated by our approach and the trees generated by C4.5, we see slight improvement in three domains: monk3, soybean-large, and tic-tac-toe. In each domain, we generated four trees using C4.5 without post-pruning, C4.5 with post-pruning, our approach with corss validation but without post-pruning the final tree, and our approach with cross validation and with post-pruning. Then, in order to compare our approach with C4.5,

we chose the tree with the best performance among the first two trees (generated using C4.5 with and without post-pruning) and the tree with the best performance among the remaining two trees (generated using our approach with and without post-pruning) and compare the generalization performance of the two chosen trees. The result was, obviously, a limited improvement over C4.5.

4.7 Discussion

This chapter introduced a new approach aiming at generating decision trees with better quality than those generated by C4.5 package. This approach is based on the idea of weighted cases which involves assigning positive real-valued weights to cases to control their contribution to the attributes' entropy scores in the attribute selection process and, in turn, the tree generation process.

The search for weight vectors that give better decision trees was conducted using the well-known strategy of simulated annealing. This approach was used to improve decision tree learning in two aspects: generating trees with less average cost of classifying a given case, and generating trees with better generalization performance.

We ran experiments for both cases and were able to generate trees with slightly less average costs than that of C4.5 in the domains: agaricus-lepiota, led-display-

creator, crx, kr-vs-kp, monk1, and soybean-large as well as trees with better generalization performances in the domains: monk3, soybean-large, and tic-tac-toe.

Overall, we can say that the reported approach added a limited improvement over C4.5.

Chapter 5

A Successive Refinement Approach to Attribute Selection

5.1 A New View to Attribute Importance

In C4.5 as well as other existing top-down decision tree learning algorithms, a decision tree is built from the cases in the training set based on a greedy, non-backtracking, non-lookahead technique. The information-gain and other similar measures used in these algorithms to evaluate attributes are not guaranteed to identify the best attribute to use. When an attribute scores well under these measures, then this attribute is usually good for splitting. However, when it scores low, then

we can not claim that it is a bad choice.

Quite often, good attributes score low and others get selected instead due to the locality of decision as discussed earlier in Chapter 3. In this case, we are stuck with these choices and their consequences.

In Chapter 4, a new approach was described based on the idea of assigning weights to cases in the training set and altering these weights in order to build new trees with better quality. The approach employs the simulated annealing strategy to guide the search in the space of weights.

However, tuning up the parameters of simulated annealing is not an easy job to do. There are no guarantees about obtaining the vector which produces a better tree by searching through a huge space like the space of vectors using a heuristic strategy like simulated annealing.

In this chapter, we propose a different iterative approach which involves studying the tests appearing in the last generated tree and then generating the next tree based on the "lessons" derived from the previous tree as well as the training examples. For each attribute, some positive real-valued *importance* score is computed based mainly on the number of cases for which the attribute plays a role during classification by the previous tree.

Let us denote the importance score of attribute A by $Impt(A)$. If during classification of a case e, a test node labelled with A is passed, then e contributes to $Impt(A)$. In contrast, cases classified without checking attribute A do not contribute to $Impt(A)$.

Under this setting, a new tree is generated considering both the importance scores as well as the information-gain scores of the attributes as illustrated in Figure 5.1. The final score of each attribute should be a function of both scores because we want to preserve the benefits of using the information-gain measure as well as taking advantage of what we learned from the tree we have at hand in building a better tree than the tree built by C4.5.

In the approach proposed here, the final attribute score for attribute A is calculated as

$$GI(A) = e^{\alpha \times Impt(A)} + e^{(1-\alpha) \times Information-Gain(A)}, \tag{5.1}$$

for some appropriately chosen α, $0 < \alpha < 1$. We will call this new measure the gain-importance measure.

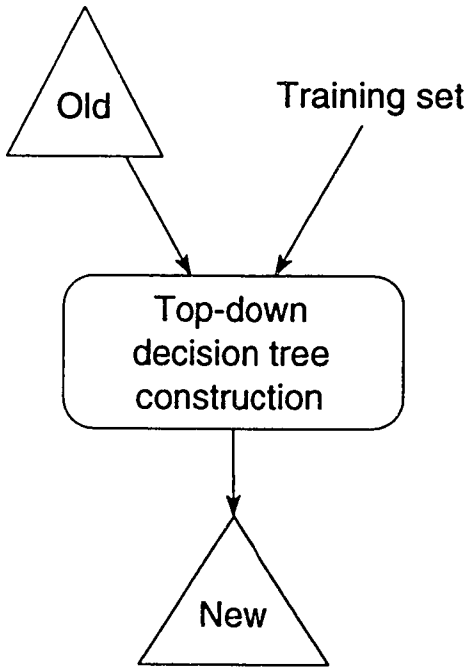


Figure 5.1: The new tree is built from the old tree and the training set

5.2 Formal Definition of the Importance Measure.

Given a tree T and an attribute A, suppose a test labelled with A lies on the path from the root of T to the leaf which gives the class of a case e in the training set. Then this case contributes to the importance score of attribute A an amount equals to the logarithm of the depth of the test node labelled with A. This amount will be denoted by $\log_2(level_A(e))$, where the level of the attribute labeling the root node is 1. Therefore, the importance score of attribute A with respect to tree T is equal to

the sum of the contributions of all the cases in the training set which is calculated as

$$\Sigma_{e \in S} \log_2(level_A(e))$$

where S is the training set.

Normalizing the importance score is necessary if we want a combined attribute score which makes use of the information-gain and the importance scores in a balanced manner. Obviously, the above score can be much greater than 1, while the information-gain score is at most 1 because the difference in entropy of the set of cases before and after splitting is at most 1. Thus, the importance score will dominate the attribute final score and we will be loosing the benefits of the information-gain score.

Therefore, we need to normalize the importance score and make it in the same range as the information-gain score. Note that the maximum value of $Impt(A)$ occurs when some attribute appears at all the bottom nodes of the current tree as is the case for attribute A in the tree shown in Figure 5.2.

However, this case rarely happens, and thus, using it as the basis for normalization results in values for $Impt(A)$ that are too small. Instead, we choose to divide the above quantity by $\mid S \mid$ times the average level of the tree. The importance score

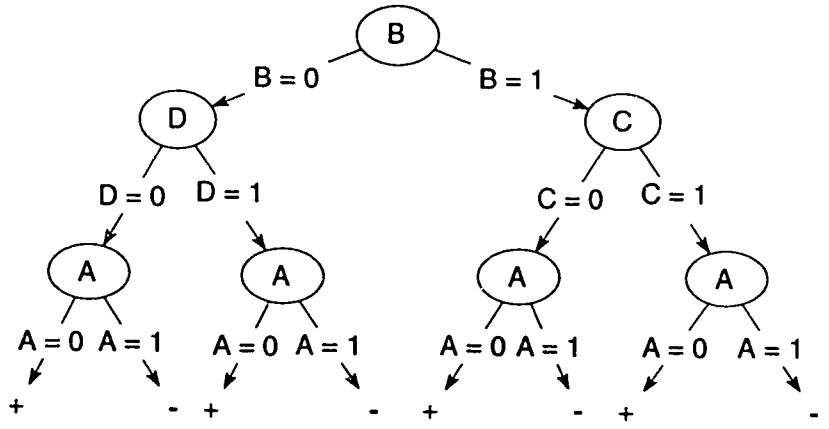


Figure 5.2: Attribute A appears at all the bottom nodes of the tree.

then becomes

$$Impt(A) = \frac{\sum_{e \in S} \log_2(level_A(e))}{|S| \times AvgLevel(T)}$$

Although this quantity may exceed 1, this possibility almost never occurs in practice. The above quantity can thus be viewed to range between 0 and 1 just as the information gain.

To illustrate the effect of computing the importance scores for the attributes and using these scores with the information-gain scores to determine which attribute to select, let us go back to the problem of learning the function $(\neg A \wedge \neg B) \vee (A \wedge \neg C)$ discussed in Chapter 3. In that problem, attribute A scored a low score under the information-gain measure, although it is the best choice at the root of the tree.

Attributes A and B scored 0.0 and 0.205, respectively, under the information-gain measure as calculated in Chapter 3. The average depth of T_0 generated by C4.5 is 2.67. Therefore, the importance scores for attributes A, B, and C are computed as follows.

$$\text{For } A: \; Impt(A) = \frac{3500 \log(2.00)}{3500 \times 2.67} = 0.706$$

$$\text{For } B: \; Impt(B) = \frac{3500 \log(1.00)}{3500 \times 2.67} = 0.000$$

$$\text{For } C: \; Impt(C) = \frac{1660 \log(3.00)}{3500 \times 2.67} = 0.531$$

As a result, attribute A scores the best importance score among the three attributes. By choosing an appropriate value for α, attribute A would score the best under the gain-importance measure and hence gets the chance to be selected earlier in the new tree.

The choice of a proper α in the gain-importance measure $GI(A)$ plays a vital role in its success as a balanced measure. A high α results in importance score domination. As a consequence, an attribute with high information-gain and low importance will not be selected if there is another attribute with low information-gain and high importance leading to a higher gain-importance score.

On the other hand, a low α will result in information-gain score domination causing this approach to degenerate to C4.5. Therefore, we would like to choose a

balanced α which preserves the power of the information-gain measure and makes use of the attribute importance score in order to learn from previous mistakes and correct wrong selections of attributes made by C4.5 in the current tree at hand.

5.3 The Successive Refinement Strategy

The approach we are proposing here implements a successive refinement strategy which exploits knowledge gained from the current tree in building a better decision tree. This successive refinement strategy works as follows. We generate the first tree T_0 using C4.5 with the information-gain criterion alone as our attribute selection criterion. Then, a new tree T_1 is built using the gain-importance measure to select attributes. For each attribute, the gain-importance score is calculated from the importance score, computed from the current tree T_0, and the information-gain score computed from the training set.

Similarly, another tree T_2 can be generated using the gain-importance measure in which importance scores are computed based on T_1. The process of generating new trees using the gain-importance measure continues in this iterative manner aiming at improving the quality of the tree.

5.4 Successive Refinement to Lower the Average Cost

As mentioned previously, the average cost is defined as the average number of tests that must be performed and satisfied before determining the class to which a case belongs. The lower the average cost value, the less number of tests needed to classify a case on average, and, in turn, classification cost becomes more economical.

The goal in this section is to employ our successive refinement technique in order to lower the average cost of the learned decision tree.

5.4.1 An Outline of the Experiment

The approach is outlined as follows:

1. Run C4.5 to form the initial tree T_0 using the information-gain attribute selection criterion and set this tree as T_{best}.

2. Run C4.5I (a modified version of C4.5 which uses the gain-importance criterion) to build T_{i+1}, where $i = 0$ initially. For each attribute, C4.5I computes the importance score (from T_i) and the information-gain score and calculates the gain-importance score. It then selects the attribute with the maximum

gain-importance score for splitting.

3. If T_{i+1} is better than T_{best} then set T_{i+1} as T_{best}.

4. If the pre-specified number of iterations is exceeded then return T_{best}. Otherwise, increment i and go to step 2.

5. Compare the average cost of the tree generated by our approach and the tree generated by C4.5.

5.4.2 Experimental Results 3

This experiment was carried out to test our approach which implements the successive refinement strategy aiming at generating more economical decision trees. It was run on several artificial and natural databases available from the UCI Machine Learning Repository. For more information about the origins, past usage, and other details of these domains, refer to appendix A.

In this experiment, we did not use domains with continuous attributes because the current implementation of our approach is unable to handle such attributes. Continuous attributes can appear more than once with different thresholds on the same path from the root to a leaf, and hence, get extra importance.

The goal in this experiment is to compare the average cost of the tree generated

by C4.5 and the tree generated by our approach. To be fair, we need to compare trees with similar degrees of consistency over the training set.

Note that we can control the degree of post-pruning by changing the value of the confidence factor, as explained in Chapter 2, in order to bring the degree of consistency of C4.5 tree to our tree's degree.

Figure 5.3 shows the results we got from running our experiment on various domains with pre-pruning and using the gain-importance criterion to select attributes. We generated the trees using C4.5 and our approach and reported the trees with similar degree of consistency.

By comparing trees with similar degrees of consistency, we have clear improvements in seven domains: led-display-creator, kr-vs-kp, monk1, monk2, monk3, soybean-large, and tic-tac-toe. Clearly, our approach proves worthy and was able to achieve the goal of obtaining more economical decision trees than those generated by C4.5 in some domains.

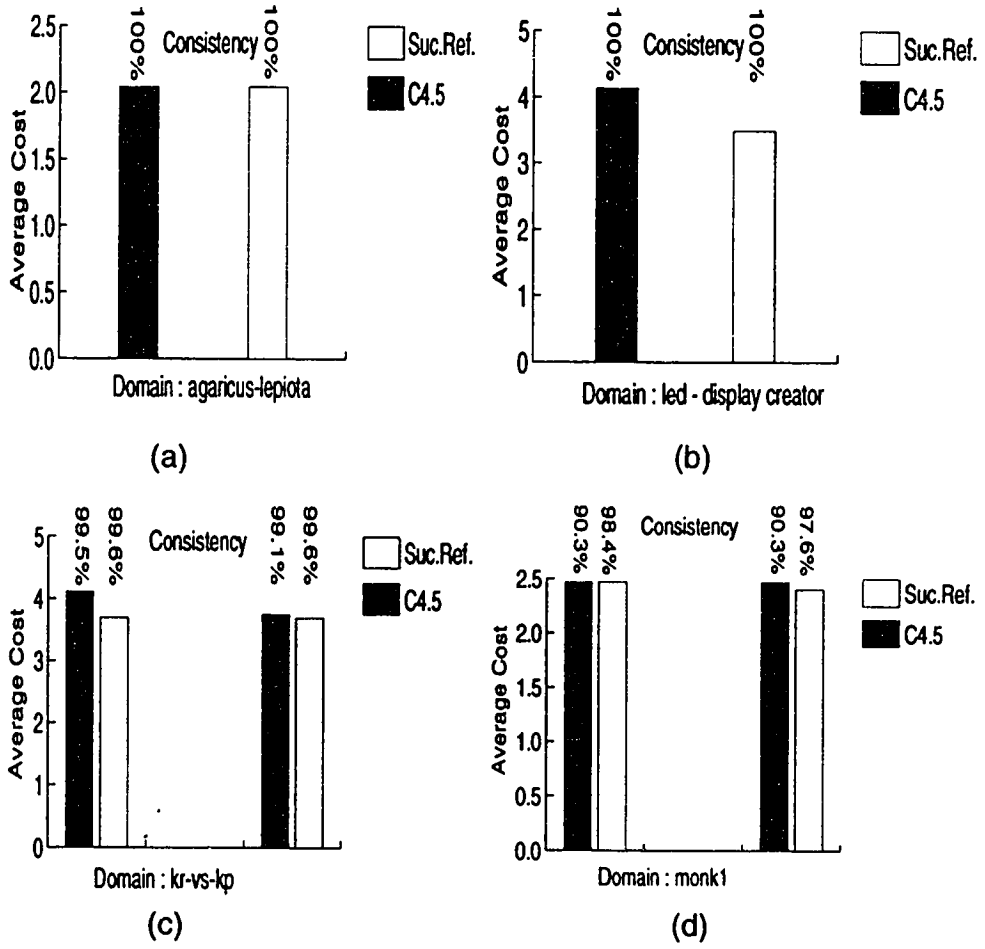
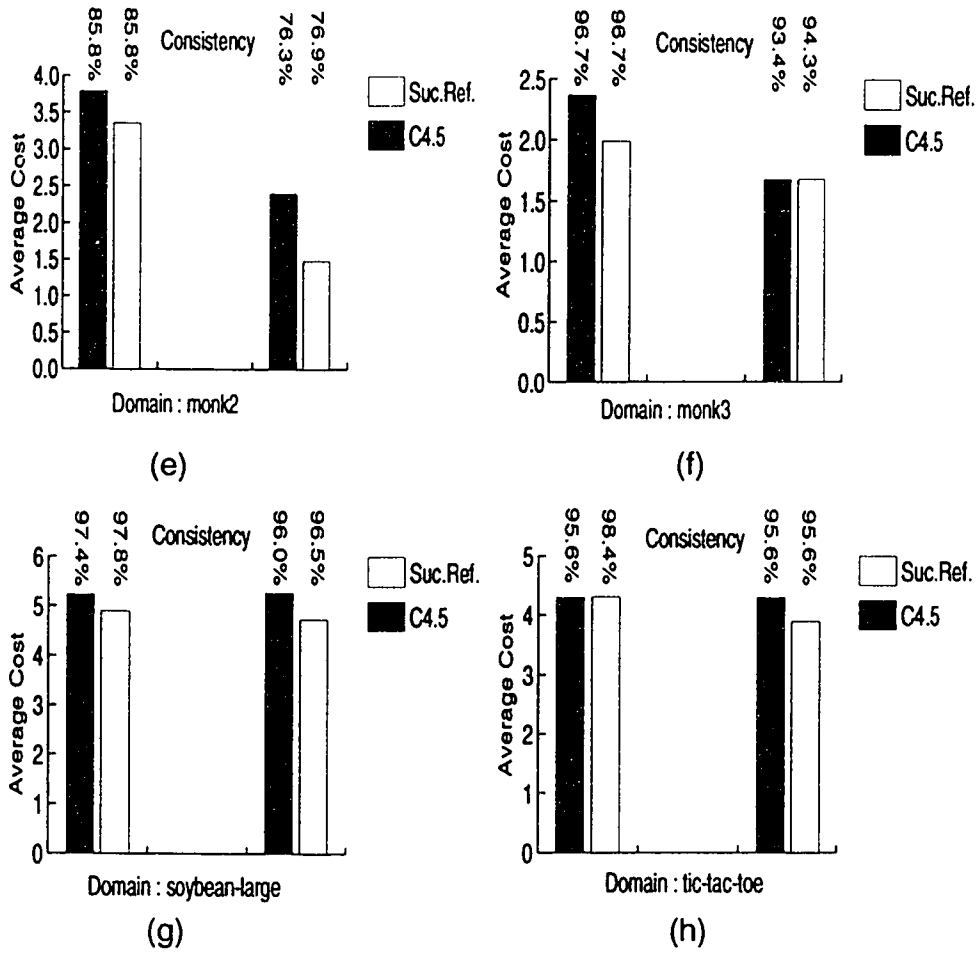


Figure 5.3: A comparison between C4.5 and the successive refinement approach in terms of the average classification cost.



5.5 Weight Adjustment to Improve the Generalization Performance

The goal of this section is to employ the proposed successive refinement technique to obtain trees with better generalization performance. As explained in Section 4.6,

the test set can not be used to select the tree which has the best generalization performance among the sequence of trees generated by our method since C4.5 is never given the chance to access the test set during the tree construction process. For this reason, we follow a cross validation strategy using the training examples alone during successive refinement.

5.5.1 Cross Validation Strategy

In this work, the test set is set aside and the cases in the training set were split into 10 blocks such that the class distribution of cases among the blocks is as uniform as possible. Each time one of the blocks is used as a tuning set and the remaining 9 blocks are used to build the tree using the successive refinement strategy, with the gain-importance attribute selection criteria, to improve the generalization performance over the tuning set.

Next time a different block is used as a tuning set and the remaining 9 blocks are used as a training set. This training set is then used, as before, to build a new tree. Finally, 10 decision trees are built, each of which has the best generalization performance over its own tuning set.

The final tree is then generated using the knowledge we learned from the 10 trees. For each attribute, there are 10 importance scores corresponding to the 10 trees. The

sum of these 10 importance scores is used for attribute selection. The attribute with the highest sum is used for splitting. This process is repeated recursively as in C4.5 to build the final tree in a top-down manner.

5.5.2 An Outline of the Experiment

Our approach can be described as follows:

1. Shuffle the cases in the training set and split them into 10 near-equal blocks at random and set the test set aside. The class distribution of the cases in each block should be as uniform as possible.

2. Choose 9 blocks to form the training set and let the remaining block be the tuning set for the current iteration.

3. Run C4.5 to build the initial tree T_0 from the training set using the information-gain attribute selection criterion and let this tree be T_{best}.

4. Run C4.5I (a modified version of C4.5 which uses the gain-importance criterion) to form T_{i+1}, where $i = 0$ initially. For each attribute, C4.5I calculates the importance score (from T_i) and the information-gain score and computes the gain-importance score. It then selects the attribute with the maximum gain-importance score for splitting.

5. If T_{i+1} is better than T_{best} then set T_{i+1} as T_{best}.

6. If the pre-specified number of iterations is exceeded then go to step 7. Otherwise, increment i and go to step 4.

7. Save the best tree generated so far with respect to the generalization performance over the tuning set.

8. If every block has already had exactly one chance to be used as the tuning set, then go to step 9. Otherwise go to step 2.

9. Restore the 10 saved trees and build the final tree from the original training set (the union of the 10 blocks) using the sum of importance scores of each attribute in the 10 trees as the criteria for attribute selection.

10. Compare the generalization performance of the final tree on the test set (which was set aside at the beginning) with the tree generated by C4.5.

5.5.3 Experimental Results 4

This experiment was carried out to evaluate the performance of our successive refinement approach with cross validation in improving the generalization accuracy compared to C4.5, following the procedure described in the previous sub-section.

The experiment was run on several artificial and natural databases available from

the UCI Machine Learning Repository. For more information about the origins, past usage, and other details of these domains, refer to appendix A.

In this experiment, we excluded domains having continuous attributes for the reasons discussed in Sub-section 5.4.2.

The results we obtained from running the experiment on various domains are shown in Figures 5.4 and 5.5. In each domain, we generated four trees using C4.5 without post-pruning, C4.5 with post-pruning, our approach without post-pruning, and our approach with post-pruning. In order to compare the proposed approach with C4.5, we chose the tree with the best performance among the first two trees (generated using C4.5 with and without post-pruning), and the post-pruned tree generated using our approach because the trees generated using our approach without post-pruning were overfitted with the training examples. The generalization performance of the two chosen trees is then compared.

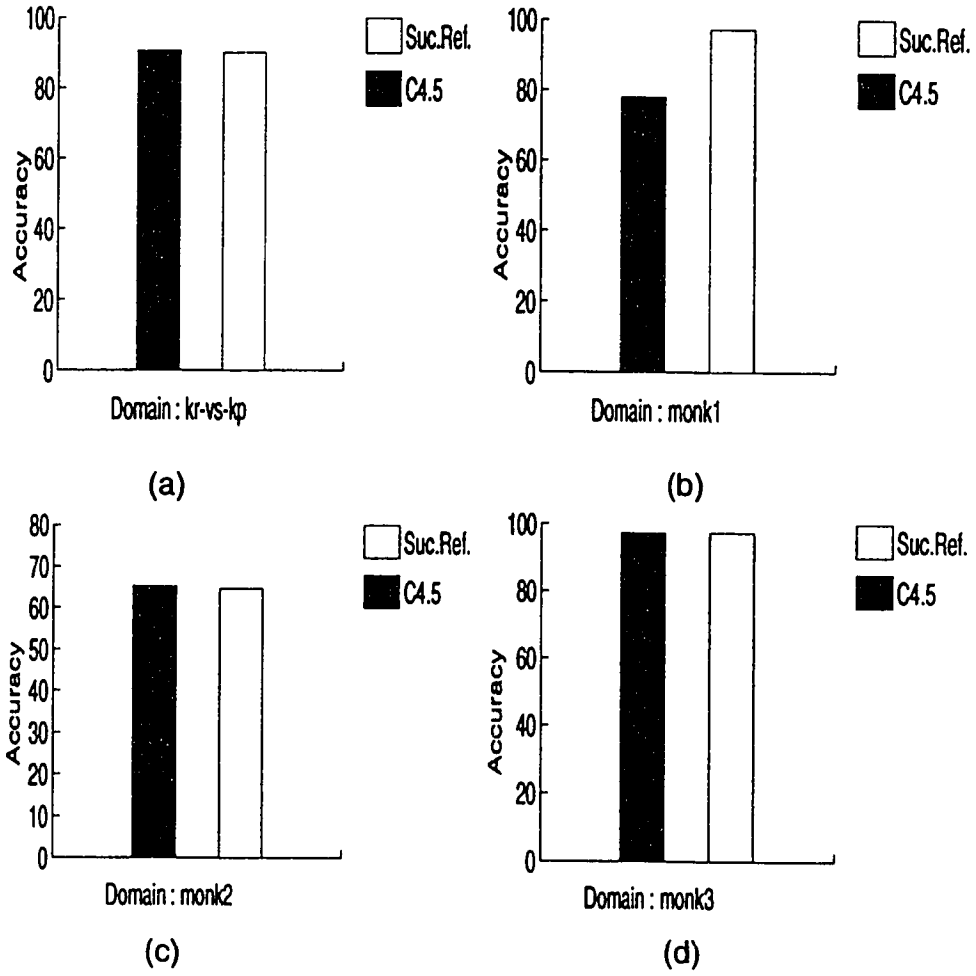
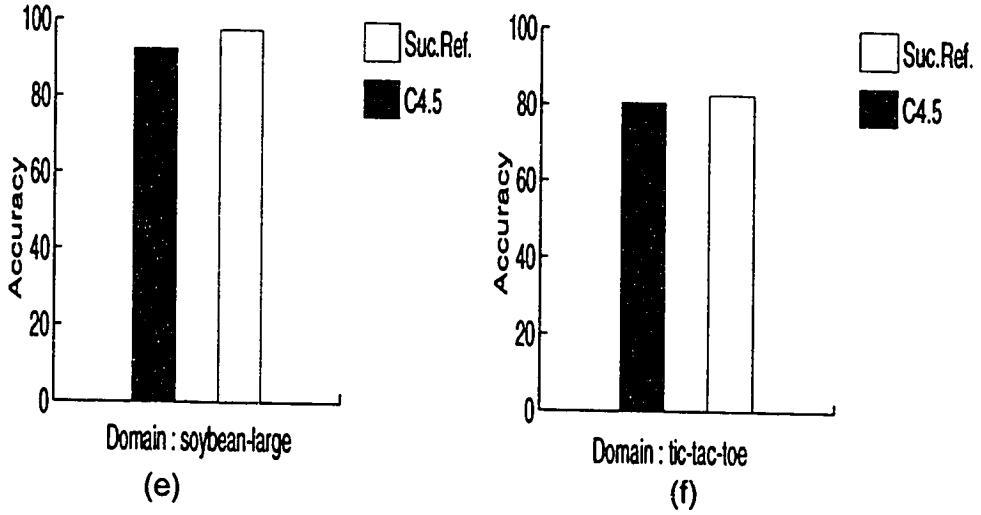


Figure 5.4: A comparison between C4.5 and the successive refinement approach in terms of the generalization performance of the tree.



As a result, the generalization performance of the trees generated by our approach has improved significantly in the domains: monk1, monk2, monk3, soybean-large, and tic-tac-toe. From these results we can conclude that our successive refinement approach is an effective technique for improving the generalization performance of learned decision trees.

5.6 Discussion

A new approach was introduced in this chapter targeted at generating better quality decision trees than those generated by C4.5 package. The approach is based on the idea of computing an importance score for each attribute based on the number of cases classified by test nodes labelled with this attribute in the current tree.

of cases classified by test nodes labelled with this attribute in the current tree. Importance scores computed from the current tree, along with the information-gain scores computed from the training examples are then used to build a new tree of better quality.

A successive refinement strategy was developed to compute the final score for each attribute in the attribute selection process. These scores are then used in the tree generation process.

Some existing algorithms follow a different approach by assigning pre-specified importance scores to the attributes based on their costs in real life. However, if all attributes have the same importance, they will be assigned the same importance score and this approach will degenerate to an approach in which the information-gain score is the only criteria used to determine which attribute to choose. In this case, the approach will suffer from the locality of decision as discussed in Chapter 3.

In contrast, our approach does not assign importance scores to certain attributes manually. The importance score of each attribute is computed internally and it depends only on the number of examples classified by the attribute.

Some experiments were performed on artificial and natural data to evaluate the ability of the proposed approach to improve decision tree learning in two aspects: generating trees with less average classification cost, and generating trees with better

generalization performance. These experiments revealed that the approach is able to generate trees with less average cost than those of C4.5 in the domains: led-display-creator, kr-vs-kp, monk1, monk2, monk3, soybean-large, and tic-tac-toe as well as trees with better generalization performance in the domains: monk1, monk2, monk3, soybean-large, and tic-tac-toe.

Overall, the experiments suggest that the approach presented in this chapter leads to better improvement over C4.5 than the approach discussed in Chapter 4.

Chapter 6

Conclusion & Future Work

Existing top-down decision tree learning algorithms are based on heuristics, and thus, they typically give only sub-optimal decision trees. Given the recent advances in computer hardware technology, more computations can now be carried out at lower costs. In real-world applications, one might be willing to spend hours and possibly days of CPU time in order to get better decision trees that suit specific requirements, e.g. better generalization performance (more accuracy in predicting the classes of new cases) and/or less average classification cost per case, than those trees generated by the rapid algorithms ID3 and CART.

To achieve this goal we propose two approaches: the weight adjustment approach and the successive refinement approach. The weight adjustment approach is based

on the idea of assigning weights to the examples and adjusting these weights slowly in an iterative manner in order to alter the scores of the attributes in the attribute selection process so that a better decision tree is generated. On the other hand, the successive refinement approach is based on a new attribute selection criterion which combines the attribute's information-gain score (as used in ID3) computed from the training examples and the importance score related to the number of examples for which the attribute plays a role during classification by the previous tree.

To evaluate these two approaches, we reported 4 experiments in which we generated decision trees from a variety of artificial and natural databases available from the Machine Learning Repository in the University of California at Irvine.

The first approach yielded only a slight improvement over C4.5, which is the most recent implementation of the ID3 algorithm, with respect to the average classification cost and the generalization performance of the decision trees. In contrast, the successive refinement approach gave more significant improvement.

For future research, one may consider using a different strategy in searching for the weight vector which produces a good tree than the simulated annealing strategy used in our approach.

Future research needs also to explore the consequences of assigning different values of α in gain-importance measure GI. In addition, measures other than GI

need to be tested considering different ways for combining of the information-gain score and the importance score.

Appendix A

Domains Used

The four experiments done in this work were run on several artificial and natural databases available from the Machine Learning Repository of the university of California, Irvine.

For each domain, there are three input files with the same filename and different extensions. First, the file with the extension "names" describes the attributes and the classes used in the domain. The file with the extension "data" holds the training set cases. Finally, the file with the extension "test" holds the test set cases.

The structure of the three input files among all the domain used for C4.5, C4.5W, C4.5I, is the same. The structure of the "names" file is as follows. Any comment line about the domain is preceded by the character —. Then, the classes of the domain are given separated by commas and ended by a full stop as shown in the following example. $+, -$.

Next, the attributes and their values are listed with each attribute followed by a colon then its values follow separated by commas and ended by a full stop as illustrated by the following example.

X1: 0, 1.
X2: continuous.
X3: ignore.

Attributes could have discrete, continuous, or ignored values. Discrete attributes have discrete values, such as, t for true, f for false, a, b, 0, 1, or any other values selected from a finite pre-specified set of values. Discrete attributes when selected to label a test node result in one branch for each possible value specified in the "names" file.

Continuous attribute have continuous values, such as, integers or reals. Continuous attributes are specified by the word "continuous" written against the attribute

followed by a full stop as in the above example. When a test node is labelled with a continuous attribute, a cut is specified so that the set of cases is partitioned in two subsets. For example, a cut could be $A > 70$ or $B \leq 45$, or similar inequalities.

Finally, ignored attribute are attributes which cannot be selected and can have any value. Usually, these attributes are used for a specific purpose, as the index attribute imposed in experiment 2.

Files with the extensions "data" and "test" also have the same structure for all the domains. The cases in these files are listed as follows. The values of the attributes, as ordered in the "names" file, are written separated by commas followed by the class to which this case belongs. This means that if the definition of attribute X1 precedes that of attribute X2, then the value of attribute X1 should precede the value of attribute X2 in every case in the training set or the test set. The following 3 cases example illustrates this point following the previous definition of attributes X1, X2, and X3. $0, 5.4, 1, +\ 1, 7.3, 2, +\ 1, 2.6, 3, -$

Furthermore, the domains we used did not have unknown values. In other words, all attributes of all cases in the training as well as the test set have values. In addition, each case's class in both sets is also known.

A description of the domains used follows.

1. **Agaricus-lepiota**

   Sources: Mushroom domain,

   (a) Mushroom records drawn from The Audubon Society Field Guide to North American Mushrooms (1981). G. H. Lincoff (Pres.), New York: Alfred A. Knopf.

   (b) Donor: Jeff Schlimmer (Jeffrey.Schlimmer@a.gp.cs.cmu.edu). Date: 27 April 1987.

   Past Usage: .

   (a) Schlimmer,J.S. (1987). Concept Acquisition Through Representational Adjustment (Technical Report 87-19). Doctoral dissertation, Department of Information and Computer Science, University of California, Irvine.

   (b) Iba,W., Wogulis,J., & Langley,P. (1988). Trading off Simplicity and Coverage in Incremental Concept Learning. In Proceedings of the 5th International Conference on Machine Learning, 73-79. Ann Arbor, Michigan: Morgan Kaufmann.

Relevant Information: This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family (pp. 500-525). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy.

Number of Instances: Training set contains 3762 cases. Test set contains 1881 cases.

Number of Attributes: 22.

Number of Classes: 2 (e=edible, p=poisonous).

2. **Thyroid Domain**

Sources: .

(a) Donors: Randolf Werner evol@uniko.uni-koblenz.de

(b) Obtained from Daimler-Benz. Date: October 1992.

Past Usage: .

(a) "Optimization of the Backpropagation Algorithm for Training Multilayer Perceptrons".

(b) "Synthesis and Performance Analysis of Multilayer Neural Network Architectures".

Relevant Information: The problem is to determine whether a patient referred to the clinic is hypothyroid. Therefore three classes are built: normal (not hypothyroid), hyperfunction and subnormal functioning. Because 92 percent of the patients are not hyperthyroid a good classifier must be significant better than 92%. These are the attributes Quinlans used in the case study of his article "Simplifying Decision Trees" (International Journal of Man-Machine Studies (1987) 221-234). Unfortunately this data differ from the version already present (donated by Ross Quinlan) I (Randolf Werner) don't know any more details about the dataset. But it's hard to train Backpropagation ANNs with it. The dataset is used in two technical reports.

Number of Instances: Training set contains 3772 cases. Test set contains 3428 cases.

Number of Attributes: 21 (15 attributes are binary, 6 attributes are continuous).

Number of Classes: 3 (1=normal (not hypothyroid), 2=hyperfunction, and 3=subnormal functioning).

3. **LED display domain**

Sources: .

(a) Breiman,L., Friedman,J.H., Olshen,R.A., & Stone,C.J. (1984). Classification and Regression Trees. Wadsworth International Group: Belmont, California.

(b) Donor: David Aha. Date: 11/10/1988.

Past Usage: .

(a) CART book.

(b) Quinlan,J.R. (1987). Simplifying Decision Trees. In International Journal of Man-Machine Studies.

(c) Tan,M. & Eshelman,L. (1988). Using Weighted Networks to Represent Classification Knowledge in Noisy Domains. In Proceedings of the 5th International Conference on Machine Learning, 121-134, Ann Arbor, Michigan: Morgan Kaufmann.

Relevant Information: This simple domain contains 7 Boolean attributes and 10 concepts, the set of decimal digits. Recall that LED displays contain 7 light-emitting diodes - hence the reason for 7 attributes. The problem would be easy if not for the introduction of noise. In this case, each attribute value has the 10% probability of having its value inverted. It's valuable to know the optimal Bayes rate for these databases. In this case, the misclassification rate is 26% (74% classification accuracy).

Number of Instances: Training set contains 3214 cases. Test set contains 1606 cases.

Number of Attributes: 24.

Number of Classes: 10.

4. **Credit Approval (crx)**

Sources: (confidential) Submitted by quinlan@cs.su.oz.au.

Past Usage: See Quinlan,

(a) "Simplifying decision trees", Int J Man-Machine Studies 27, Dec 1987, pp. 221-234.

(b) "C4.5: Programs for Machine Learning", Morgan Kaufmann, Oct 1992.

Relevant Information: This file concerns credit card applications. All attribute names and values have been changed to meaningless symbols to protect confidentiality of the data. This dataset is interesting because there is a good mix of attributes - continuous, nominal with small numbers of values, and nominal with larger numbers of values. There are also a few missing values.

Number of Instances: Training set contains 465 cases. Test set contains 200 cases.

Number of Attributes: 15.

Number of Classes: 2 (+/−).

5. **Chess End-Game – King Rook versus King Pawn**

Sources: .

(a) Database originally generated and described by Alen Shapiro.

(b) Donor/Coder: Rob Holte (holte@uottawa.bitnet). The database was supplied to Holte by Peter Clark of the Turing Institute in Glasgow (pete@turing.ac.uk). Date: 1 August 1989.

Past Usage: .

(a) Alen D. Shapiro (1983,1987), "Structured Induction in Expert Systems", Addison-Wesley. This book is based on Shapiro's Ph.D. thesis (1983) at the University of Edinburgh entitled "The Role of Structured Induction in Expert Systems".

(b) Stephen Muggleton (1987), "Structuring Knowledge by Asking Questions", pp.218-229 in "Progress in Machine Learning", edited by I. Bratko and Nada Lavrac, Sigma Press, Wilmslow, England SK9 5BB.

(c) Robert C. Holte, Liane Acker, and Bruce W. Porter (1989), "Concept Learning and the Problem of Small Disjuncts", Proceedings of IJCAI. Also available as technical report AI89-106, Computer Sciences Department, University of Texas at Austin, Austin, Texas 78712.

Relevant Information: The pawn on a7 means it is one square away from queening. It is the King Rook's side (white) to move.

Number of Instances: Training set contains 1315 cases. Test set contains 1888 cases.

Number of Attributes: 36.

Number of Classes: 2 (*won*=White can win, *nowin*=White cannot win).

6. **The Monk's Problems**

Relevant Information: The MONK's problems are a collection of three binary classification problems over a six-attribute discrete domain. Each training/test case is of the form:

$$< name >:< value1 >, < value2 >, < value3 >, < value4 >,$$

$$< value5 >, < value6 >, < class >$$

where $< name >$ is an ASCII-string, $< valuen >$ represents the value of attribute # n, and $< class >$ is either 0 or 1, depending on the class this example belongs to. The *true* concepts underlying each MONK's problem are given by:

(a) MONK-1: $(attribute_1 = attribute_2)$ or $(attribute_5 = 1)$.

(b) MONK-2: $(attribute_n = 1)$ for EXACTLY TWO choices of n (in $\{1, 2, ..., 6\}$).

(c) MONK-3: $(attribute_5 = 3$ and $attribute_4 = 1)$ or $(attribute_5 \neq 4$ and $attribute_2 \neq 3)$.

Number of Instances: .

(a) For MONK-1: Training set contains 124 cases. Test set contains 432 cases.

(b) For MONK-2: Training set contains 169 cases. Test set contains 432 cases.

(c) For MONK-3: Training set contains 122 cases. Test set contains 432 cases.

Number of Attributes: 6.

Number of Classes: 2.

7. **Large Soybean Database**

Sources: .

(a) Michalski,R.S. Learning by being told and learning from examples: an experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis, International Journal of Policy Analysis and Information Systems, 1980, 4(2), 125-161.

(b) Donor: Ming Tan & Jeff Schlimmer (Jeff.Schlimmer@cs.cmu.edu). Date: 11 July 1988.

Past Usage: .

(a) See above.

(b) Tan, M., & Eshelman, L. (1988). Using weighted networks to represent classification knowledge in noisy domains. Proceedings of the Fifth International Conference on Machine Learning (pp. 121-134). Ann Arbor, Michigan: Morgan Kaufmann.

(c) Fisher,D.H. & Schlimmer,J.C. (1988). Concept Simplification and Predictive Accuracy. Proceedings of the Fifth International Conference on Machine Learning (pp. 121-134). Ann Arbor, Michigan: Morgan Kaufmann.

Relevant Information: There are 19 classes, only the first 15 of which have been used in prior work. The folklore seems to be that the last four classes are unjustified by the data since they have so few examples. There are 35 categorical attributes, some nominal and some ordered. The value "dna" means does not apply. The values for attributes are encoded numerically, with the first value encoded as "0," the second as "1," and so forth. An unknown values is encoded as "-1."

Number of Instances: Training set contains 227 cases. Test set contains 40 cases.

Number of Attributes: 35.

Number of Classes: 19.

8. **Tic-Tac-Toe Endgame database**

Sources: .

(a) Creator: David W. Aha (aha@cs.jhu.edu)

(b) Donor: David W. Aha (aha@cs.jhu.edu). Date: 19 August 1991

Past Usage: .

(a) Matheus, C. J., & Rendell, L. A. (1989). Constructive induction on decision trees. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 645-650). Detroit, MI: Morgan Kaufmann.

(b) Matheus, C. J. (1990). Adding domain knowledge to SBL through feature construction. In *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 803-808). Boston, MA: AAAI Press.

(c) Aha, D. W. (1991). Incremental constructive induction: An instance-based approach. In *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 117-121). Evanston, ILL: Morgan Kaufmann.

Relevant Information: This database encodes the complete set of possible board configurations at the end of tic-tac-toe games, where "x" is assumed to have played first. The target concept is "win for x" (i.e., true when "x" has one of 8 possible ways to create a "three-in-a-row"). Interestingly, this raw database gives a stripped-down decision tree algorithm (e.g., ID3) fits. However, the rule-based CN2 algorithm, the simple IB1 instance-based learning algorithm, and the CITRE feature-constructing decision tree algorithm perform well on it.

Number of Instances: Training set contains 912 cases. Test set contains 46 cases.

Number of Attributes: 9.

Number of Classes: 2.

Bibliography

[AAK95] H. Almuallim, Y. Akiba, and S. Kaneda. On Handling tree-structured attributes in decision tree learning. In *The Proceedings of the 12th International Conference on Machine Learning*, July 1995.

[AAK96] H. Almuallim, Y. Akiba, and S. Kaneda. An Efficient Algorithm for Finding Optimal Gain-Ratio Multiple-Split Tests on Hierarchical Attributes in Decision Tree Learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, August 1996.

[Ber90] F. Bergadano. Guiding Induction with Domain Theories. In *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 474–492. Morgan kaufman, 1990.

[BFOS84] L. Breiman, J. H. Freidman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Belmont, Wadsworth, 1984.

[BR88] L. Blum and R. L. Rivest. Training a 3-node Neural Network is NP-Complete. In *Proceedings of the First ACM Workshop on Computational Learning Theory*, pages 9–18, 1988.

[Bro95] C. Brodley. Automatic selection of split criterion during tree growing based on node location. In *The Proceedings of the 12th International Conference on Machine Learning*, pages 73–80, July 1995.

[Cat91] J. Catlett. Overpruning Large Decision Trees. In *12th International Joint Conference on Artificial Intelligence*, pages 764–769, August 1991.

[CN89] P. Clark and T. Niblett. The CN2 Induction Algorithm. *Machine Learning*, 3:261–283, 1989.

[Coc87] J. R. B. Cockett. Discrete Decision Theory: Manipulations. *Theoretical Computer Science*, 54:215–236, 1987.

[EMS93] F. Esposito, D. Malerba, and G. Semeraro. Decision Tree Pruning as a Search in the State Space. In *European Conference on Machine Learning*, pages 165–84, April 1993.

[Fay91] U. Fayyad. *On the Induction of Decision Trees for Multiple Concept Learning.* PhD thesis, University of Mitchigan, AnnArbor, 1991.

[FI92] U. Fayyad and K. Irani. The Attribute Selection Problem in Decision Tree Generation. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 104–110, July 1992.

[HMS66] E. B. Hunt, J. Martin, and P. J. Stone. *Experiments in Induction.* New York: Academic Press., 1966.

[Hol95] L. B. Holder. Intermediate Decision Trees. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1056–1062, August 1995.

[HR76] L. Hyafil and R. L. Rivest. Constructing Optimal Binary Decision Trees is NP-Complete. *Information Processing Letters*, 5:15–7, 1976.

[KGJV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–80, May 1983.

[Kod90] Y. Kodratoff. Learning Expert Knowledge by Improving the Explanations Provided by the System. In *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 433–465. Morgan kaufman, 1990.

[KR87] S. A. Kravitz and R. A. Rutenbar. Placement by Simulated Annealing on a Multiprocessor. *IEEE Transactions on Computer-Aided Design*, CAD-6:534–49, July 1987.

[LHS95] C. K. Y. Lin, K. B. Haley, and C. Sparks. A Comparative Study of Both Standard and Adaptive Versions of Threshold Accepting and Simulated Annealing Algorithms in Three Scheduling Problems. *European Journal of Operational Research 83*, pages 330–346, 1995.

[Lub95] D. Lubinsky. Increasing the performance and consistency of classification trees by using the accuracy criterion at the leaves. In *The Proceedings of the 12th International Conference on Machine Learning*, pages 371–377, July 1995.

[LV91] J. H. Lin and J. S. Vitter. Complexity Results on Learning by Neural Nets. *Machine Learning*, pages 211–230, 1991.

[LW94] W. Z. Liu and A. P. White. The Importance of Attribute Selection Measures in Decision Tree Induction. *Machine Learning*, 15:15–41, 1994.

[MKSB93] S. Murthy, S. Kasif, S. Salzberg, and R. Beigel. OC1: Randomized Induction of Oblique Decision Trees. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 322–327, July 1993.

[MP94] P. M. Murphy and M. J. Pazzani. Exploring the Decision Forest: An Emperical Investigation of Occam's Razoring Decision Tree Induction. *Journal of Artificial Intelligence Research 1*, pages 257–275, 1994.

[MR89] C. J. Matheus and L. A. Rendell. Constructive Induction on Decision Trees. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 645–650, 1989.

[MS95a] S. Murthy and S. Salzberg. Decision Tree Induction: How Effective is the Greedy Heuristics? In *Proceedings of the first International Conference on Knowledge Discovery and Data Mining*, pages 222–227, 1995.

[MS95b] S. Murthy and S. Salzberg. Lookahead and Pathology in Decision Tree Induction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1025–1031, August 1995.

[Nor89] S. W. Norton. Generating Better Decision Trees. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 800–805, 1989.

[Pag89] G. Pagallo. Learning DNF by Decision Trees. *Machine Learning*, pages 639–644, 1989.

[PD95] L. Painton and U. Diwekar. Stochastic Annealing for Synthesis Under Uncertainty. *European Journal of Operational Research 83*, pages 489–502, 1995.

[PH90] G. Pagallo and D. Haussler. Boolean Feature Discovery in Empirical Learning. *Machine Learning*, 5:71–99, 1990.

[QR89] J. R. Quinlan and R. L. Rivest. Inferring Decision Trees Using the Minimum Description Length Principle. *Information and computation*, 80:227–248, 1989.

[Qui86] J. R. Quinlan. Induction of Decision Trees. *Machine Learning*, pages 81–106, 1986.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[Rut89] R. A. Rutenbar. Simulated Annealing Algorithms: An Overview. *IEEE Circuits and Devices Magazine*, pages 19–26, January 1989.

[Sch95] C. Schaffer. Overfitting Avoidance as Bias. In *12th International Conference on Machine Learning*, pages 1–36, July 1995.

[SFU89] S. Spangler, U. M. Fayyad, and R. Uthurusamy. Induction of Decision Trees from Inconclusive Data. In *Proceedings of the Sixth International WorkShop on Machine Learning*, pages 146–150, 1989.

[WI91] S. M. Weiss and N. Indurkhya. Reduced Complexity Rule Induction. In *12th International Joint Conference on Artificial Intelligence*, pages 678–684, August 1991.

Vitae

- Muhammad Nauwar Ahmad Nazir Jalal Al-Afandi.
- Born in 1970 at Idlib, Syria.
- Received Bachelor of Science degree (**B.Sc**) in Computer Science degree in 1992 from King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia.
- Started working in the industry right away after graduation. Started working in business applications development. At the same time, was studying for the Master in Computer Science as a part-timer.
- Received the Master of Science degree (**M.S.**) in Computer Science in July, 1996 from King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia.
- Address: Syria, Idlib, P.O. BOX 75